A Look at Security Technologies

Granite Key, Inc.

Aug 22, 2005

Granite Key has helped Fortune 100 customers integrate security technologies into their new and existing PC, Wireless, Network, and Internet solutions. Our customers typically are adding authentication, platform security, encryption, auditing into their applications. Our philosophy is that a good technical solution must start with a deep understanding of business requirements, business processes, and competing solutions, followed by design, implementation, and in many cases process engineering/re-engineering. Thus, GraniteKey frequently gets involved with customers at the earliest stages of a project, in order to help ensure the business success of the solution

Contents

Summary

This paper will provide a brief overview of various security technologies and architectural considerations for integrating, into your solutions, security technology such as authentication, encryption, auditing, document integrity and signatures. It is recommended to first read the paper "Building a Whole Security Solution". The term "wholution" is covered in this paper. It refers to the concept of the need to understand all factors (in addition to security technologies) that impact the success of your security solution, including the people, procedures, management, and measurement of success / failures.

Encryption Technology

Symmetrical Encryption

Symmetric encryption has been around for a very long time – long before computers existed. One uses symmetrical encryption to encrypt data which is then sent to a recipient who then decrypts the data. It's called symmetrical, because both sides share something secret (i.e,. a password/private key). They both have the same information. This of course implies that they know each other directly or indirectly. This might be something like a codebook, or in modern times it is a key. Today the key would typically be 64 – 256 bytes of data.

The most common symmetrical encryption technologies in use today are DES, 3DES (triple DES), and AES. DES and 3DES have been around for a long time (3DES is still in wide use today), and AES is relatively new. AES is designed to run faster than 3DES. Even though there are differences in the details of the implementation of symmetrical technologies (and some are more robust such as blowfish), they all basically work in a similar way. Data is encrypted with a key and that key must be shared with others. The length of the key generally determines the level of security. Some algorithms are more secure than others, but those which are readily available are typically secure enough, where security holes are more likely to be in other parts of the Wholution, not the particular symmetrical algorithm being used. In reality, differences in choice of a symmetrical encryption technology are typically more related to performance and cost, then in level of security.

Historically, encryption technology was a specialized, archaic technology, understood by and available only to a few. Much of it was symmetrical technology and various variations. However, recently symmetrical encryption as well as most cryptographic technology, is widely available in an open forum. The challenge is in the Wholution, not the encryption.

Public key cryptography (Asymmetrical encryption)

The differences between symmetrical and asymmetrical encryption is best described with a table:

Symmetrical	Asymmetrical
The same key is used for encryption and decryption	Keys come in pairs, if you encrypt with one key you decrypt with the other, and vice versa
All keys are secret (private)	One of the keys is secret (private) and one is available to everybody (public)
Relatively fast, can run on almost any piece of hardware.	Very slow. Runs OK on a Pentium, but not on smaller processors.
Can encrypt large quantities of data	Used with small quantities of data (e.g. 256 bytes or less)
Generally used to encrypt data	Generally use to validate identities, ensure data has not changed, and to exchange symmetrical keys between 2 or more parties

One might other than the fact that asymmetrical encryption runs slower and can't encrypt much data, why use it? The key difference is that you can use encryption technology without having to know the other party and ensure that they have access to a private key which you both share. This is especially important in a public environment (e.g. email), where you would have to share a key with hundreds if not thousands of people, each with a different set of keys.

The most common public key algorithms are RSA and Elliptic Curve. RSA is more widely used, but Elliptic Curve is considered to be faster (or more secure for the same amount of processing power used).

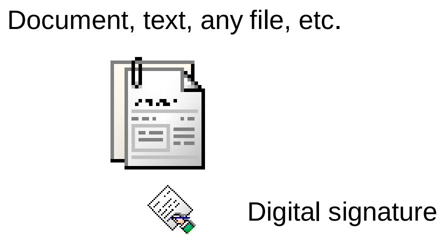
So what can you do with public key cryptography?

Applications of Public Key Cryptography

Digital Signatures and identities (with public key cryptography)

You can encrypt data with your private key and others decrypt it with your public key, which is available to everybody. So what? Well, you're the only one who has your private key (which is hopefully protected in hardware with a pin or some other mechanism). This proves that YOU encrypted the data. Two useful things you can do with this are digital signatures and proving your identity:

Digital Signature



A digital signature is typically attached to the end of a message or accompanies a document. It is similar to a written signature which indicates that this item came from you. However, it does several things that a traditional written signature does not: it ensures that the data has not been changed, and your signature cannot be copied to a different document. It works by running the document through something called a Hash function, which is used to generate a series of bytes (e.g. 20 bytes) based on the data. (An example of a trivial hash function is one that counts the number of words in the document.). SHA1 is a typical hash function. The hash function is complex enough where it is virtually impossible to figure out how to change the data to create the same hash (for example changing the document then putting a bunch of "garbage" at the end to get the same hash). This hash is then encrypted with your private key. The recipient decrypts the hash you sent with your public key, hashes the document themselves, and makes sure that the hash that he calculated matches the hash that you sent. If the hashes match, then the recipient knows that you sent the document and that it hasn't been changed.

Proving your identity

Inherent in a digital signature is proving your identity. If somebody sent you some data, and asked you to sign it, they could validate that it was you. And if they changed the data each time, it would be impossible for anyone who happened to overhear the conversation to replay the data and masquerade as you. It proves that you have your private key without revealing that key. This is similar to validating somebody's identity by asking them a series of questions, and different questions each time (for example if you think they are an imposter, you might ask questions such as, "What's your uncle's name?" or "What was your first pet's name?") However, of course, public key cryptography is much more robust in that there are trillions of questions and you don't have to know the person personally (or reveal anything of a personal nature).

Other applications of digital signatures

Digital signatures are useful for many other types of applications which attest to the fact that a document or piece of data has not been modified. For example:

- Ensuring a financial transaction has been approved by the holder of a card
- Ensuring software came from a particular vendor
- Approving documents
- Approving medical procedures
- Ensuring an email came from a particular person

Encrypting data without needing to share a secret: key exchange (with public key cryptography)

Someone can encrypt data with your public key (which is available to everybody) and you can decrypt it with your private key. Thus, anyone can encrypt data and send it to you, and *only you* can decrypt it. This is much better than symmetrical encryption because the sender doesn't have to know you (i.e., they don't have to exchange a secret key with you over the phone, in person, or through the mail). The only problem is that asymmetrical encryption runs very slowly. Solution: the sender can encrypt data using symmetrical encryption (e.g. 3DES) using a randomly generated key (symmetrical encryption runs much faster), then use your public key to encrypt the 3DES key and send it to you along with the encrypted data. And this can be done either in real time (for example while exchanging data via a website), or offline (e.g. via email or storing files). You can also send the data to multiple people by encrypting the data once with a 3DES key, and then encrypting the 3DES key for each person you want to receive the data (using their public key). The encrypted key is typically 256 bytes or less, so the impact on the amount of data transmitted is small. This symmetrical key is typically called a session key, because it is only valid only for a particular session or a particular set of data.

There is still a potential problem. Theoretically, someone can snoop around and record all the data being transmitted, then masquerade as you, by just replaying it. If they are clever, and the person who designed the system has not taken precautions, they might get something out of it. A simple technique to avoid this is anti-replay or salting. What if every time you sent data, you corrupted the data (e.g. using a commonly available XOR function) by adding a little bit of random data to it, but you send that random data (encrypted of course) to the recipient who would use that information to uncorrupt the data. This way, the same data would not be sent twice, even if the content was the same. This is enough to really confuse and thwart even a hacker with a superfast computer.

Authentication and key exchange, example SSL (with public key cryptography)

In some cases, one needs to both verify identity and encrypt data. When you goto a secure website (https instead of http, the word "secure" may appear on the webpage), you're concerned about 2 things:

1. Is this website who they claimed to be (e.g. is this really Amazon.com I'm giving my credit card information?).
2. All data to and from this website should be encrypted.

This combines both of the above described public key technologies. The transaction is complicated using a technology called SSL, but it basically works as follows:

- Your web browser challenges the website (i.e., sends it random data)
- The website responds with a digital signature and your browser validates the digital signature and warns you if it fails
- Your browser generates a random session key, encrypts it with the web server's public key and sends the web server that encrypted key
- Your browser and the web server encrypt all data sent between you using that session key.

There is only one little problem. How do you get somebody's public key, how do you know it's theirs, and how do you know it's valid?

PKI (Public Key infrastructure)

Typically you get somebody's public key in one of 2 ways:

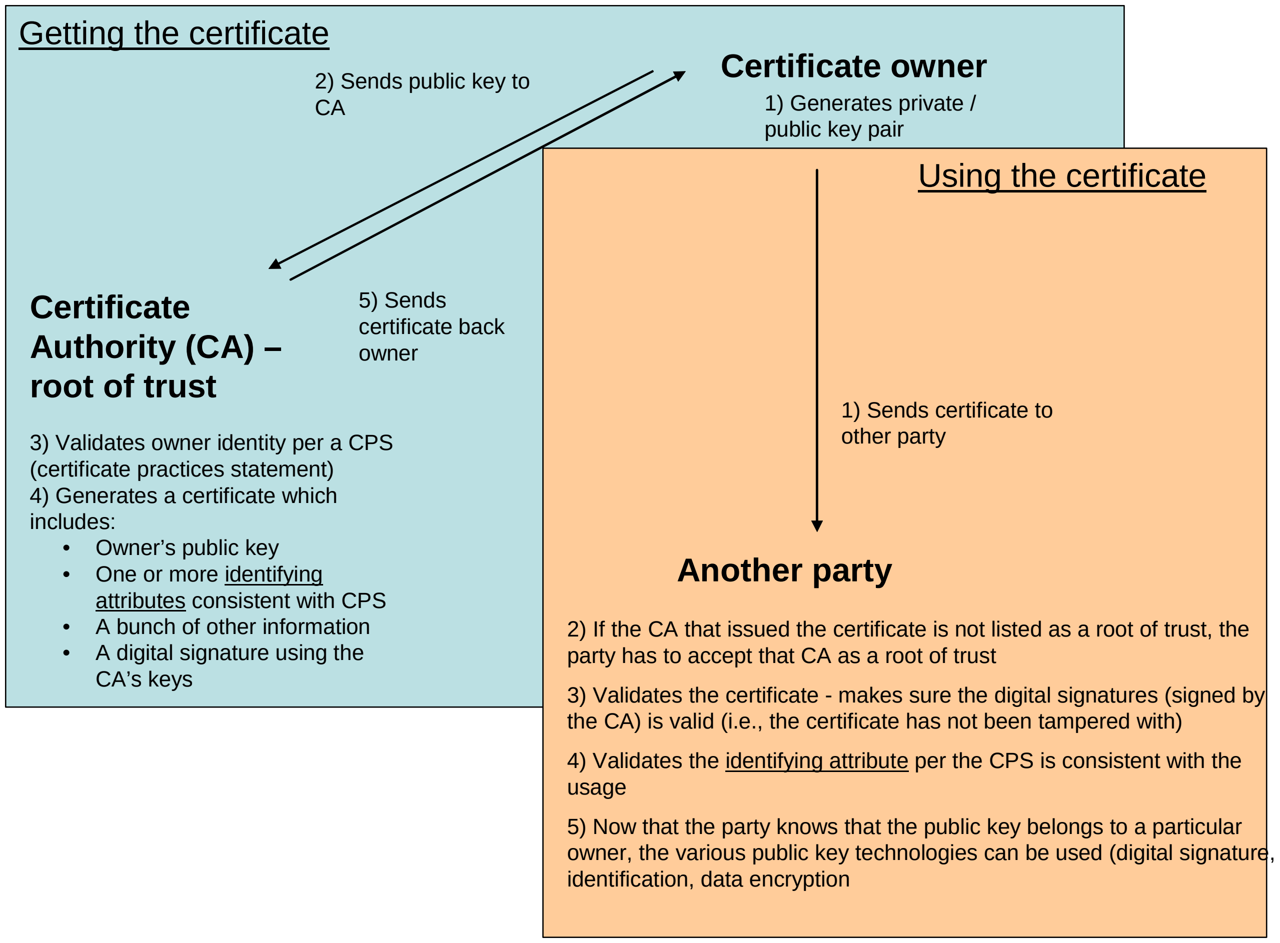
- They give it to you
- You go to a directory to get it.

The most common way is for them to give it to you. Examples:

- When connecting to a web server, the key is sent to you at the beginning of the transaction.
- When you receive an email from somebody, their key can be attached and you can save it.
- When connecting to any server or device, they can send the key to you.

This might seem ridiculous -- somebody you don't know gives you their key, and you're supposed to trust them? How do you know they are, in fact, who they say they are?

Well first of all, they don't just give you their public key, they actually give it to you in the form of a certificate, and this certificate is issued by a Certificate Authority.


Getting the certificate
Certificate owner
1) Generates private / public key pair
2) Sends public key to CA
Certificate Authority (CA) – root of trust
5) Sends certificate back owner
3) Validates owner identity per a CPS (certificate practices statement)
4) Generates a certificate which includes:
• Owner's public key
• One or more identifying attributes consistent with CPS
• A bunch of other information
• A digital signature using the CA's keys
Using the certificate
1) Sends certificate to other party
Another party
2) If the CA that issued the certificate is not listed as a root of trust, the party has to accept that CA as a root of trust
3) Validates the certificate - makes sure the digital signatures (signed by the CA) is valid (i.e., the certificate has not been tampered with)
4) Validates the identifying attribute per the CPS is consistent with the usage
5) Now that the party knows that the public key belongs to a particular owner, the various public key technologies can be used (digital signature, identification, data encryption

(Please refer to numbers on above diagram)

Getting the certificate:

1. The certificate owner (the person, entity, or device who wants to provide a public key) generates a key pair, ensuring that the private key is protected. (e.g the user requires a pin to access it, and it might be stored inside a hardware device such as a smart card)

2. The owner sends only the public key to the Certificate Authority (CA).

3. The Certificate Authority (e.g. Verisign, Microsoft), first has to establish a scheme by which you are identified. They can't use your name – there could be a million John Smith's. So they pick something that is uniquely yours and readily publicly available. For example, your email address. This is the Identifying Attribute. However, they still don't know it's you. So they ask you for your email address when you send your public key, then they send you an email, and ask you to respond to it, to validate that it's you. This methodology is reasonably secure, but more importantly it's very easy to do. This process is documented per a CPS (Certificate Practices Statement). It basically says that I will issue a certificate to you, and in doing so, I'm going to make sure it's you by

sending you an email then putting your email in the certificate along with your public key, then digitally signing the whole thing. This particular way of doing things is called a "Type 1 certificate". It's sometimes called a digital id or a personal certificate. The most important thing is that this can only be used for email. Why? Because the validation mechanism was email. If you want to send somebody encrypted data (by using their public key), you can use their certificate to get the public key, and that certificate asserts that the public key belongs to the person at a particular email address. So if you know somebody well enough to know them by email, that's probably good enough. It's not the most secure mechanism available, but it works very well, provided you know the limitations. (i.e., you understand the CPS).

There are other types of certificates. For example a "Type 3 certificate" is typically issued to the owner of a website. This is used for a secure web connection. The CPS is more complicated. The identifying attribute of the certificate is the domain (e.g. www.amazon.com), and the CPS to ensure that the company Amazon is getting the certificate and not some hacker, is more complex than just sending them an email. You basically have to do some research, involving investigating who owns the domain (e.g. the company amazon.com) and making sure that your contact at the company actually works there, etc. It's a much more complicated process, but the idea is that since you trust a company like Verisign to do this properly, you assume it's being done properly. They have a huge incentive to make sure it's done probably because if it isn't, they could go bankrupt very quickly (i.e., the importance of branding in the security business).

Type 1 and Type 3 certificates are issued by a public CA. This means that the use of the certificate is well defined for a very broad audience. There are also private CA's which issue certificates based off specific information that is relevant only to a particular group of people. For example, when issuing certificates for a VPN (i.e., within a company), the identifying attribute might be the employee's name, and that employee might be validated (i.e., providing who they are, by having them physically meet with an IT administrator, for example). If using it for a credit card, it might be the credit card number.

4. Once the identifying attribute is defined and validated, a certificate is created which includes the public key, the identifying attribute, the expiration date, and other information. The CA uses their private key to digitally sign the certificate. The CA is called a root of trust, because one trusts them implicitly. (It's like you might hire somebody who is recommended by a friend of yours. Even if you don't know the prospective employee, your friend vouches for him, and you trust your friend based on your previous relationship with him or his reputation). The CA's public key is commonly known (e.g. many CA's keys come embedded in Internet Explorer – just look under Tools / Internet Options / Content / Certificates / Trusted Root Certification Authorities). Note that the CA must be extremely careful to protect their private key, because if this key is compromised it would invalidate millions of certificates and cause much damage to the reputation and stock value of the company.

The process is similar for a private CA. However, the root key is not commonly known, it must provided by whomever has delivered the application to the customer.

5. The CA sends the certificate back to the owner. At this point, the certificate is public information so it doesn't have to be secure. Only the owner of the certificate can prove it's theirs because only they have the private key which is associated with the public key in the certificate.

Using the certificate:

1. The other party connects to the certificate owner, who then sends the certificate to the them
2. The other party extracts the CA information (embedded in the certificate) and ensures that this CA is accepted as a root of trust. If not, the user is given the opportunity to add that CA as a root of trust. It's rare that a CA needs to be added. There are only a few public CA's, most of which are already included with Windows. With a private CA, whoever delivered the application to the customer typically sets this up.
3. The other party validates the certificate's digital signature (i.e., makes sure that the certificate has not been tampered with, by validating it with the CA's commonly known public key).
4. The other party ensures that the identifying attributes is consistent with how the certificate is being used. In the case of email (Type 1 certificate), it makes sure that the identifying attribute matches the email where the data is being <u>sent</u> (in the case of encrypting data to be sent to another party) OR that the identifying attribute matches the email from whom the data is being <u>received</u> (in the case of validating a digital signature).

   In the case of https/secure web (Type 3 certificate), the application makes sure the URL is consistent with the certificate. For other applications (e.g. credit cards, proprietary applications), the nature of the application will determine how this is done.

5. Once the public key has been validated as belonging to the appropriate party it can be used. Sometimes the other party is immediately challenged to see if they have the private key (e.g. for secure web), which proves they own the certificate (i.e., proves their identity). In other cases, for example when encrypting data, the data is simply encrypted using their public key from the certificate. If the party who send you the certificate doesn't really own that certificate, it doesn't matter if you send them encrypted data without first verifying that they have the private key (proving their identity), because if they don't, they couldn't read the data anyway.

PKI involves an <u>unbroken chain of trust</u> to know who a public key belongs to, how that party is identified, and that the key is still valid. PKI is a very complex subject, and there is quite a bit more to it than is illustrated here. For example certificates can be revoked; there are various protocols for communicating data between entities. There are a lot of resources available on the web for further information.

Secure Timestamps

Digital signatures can be used to sign data which contain timestamps (as to when the transaction was completed or the data was signed), but how does one know that the timestamp is valid. Secure timing can come from a secure source (e.g., a trusted server), or one could simply make the assumption that the user checked the timestamp before the data was signed. However, there are cases where the user could purposely create a digital record using a timestamp previously generated (e.g. backdating something). Thus the entire record has to be digitally signed from a trusted source to be truly accurate (an electronic version of a notary).

Timestamps can come from device (e.g. credit card terminal), but technologies to provide securing timing in an embedded device are not typically available (vs. coming from a trusted server).

Digital timestamps have not yet hit the mainstream because in order to do so, legal issues will have to be resolved (i.e., the weak link in the Wholution. Without legal precedent the timestamps would mean nothing in a public environment). This technology will continue to evolve.

Crypto API's

In the past, cryptographic algorithms were not only complex to develop, but complex to use. The developers had their own way of structuring the interface. However, structurally, at the highest level, most cryptographic algorithms are quite simple, and share many similarities (e.g. symmetrical encryption requires a key, data to be encrypted, and a few configuration parameters regardless of the specific algorithm). Fortunately, crypto API's have come along to create a common interface across cryptographic algorithms. This makes it easier for application developers (i.e., those using the cryptographic functions, not those creating them) to use cryptographic algorithms from any vendor. Crypto algorithms have become somewhat of a commodity (focus is in speed and price) as the key security issues are in the Wholution design. Modern cryptographic algorithms are no longer the weak link. They will continue to provide adequate security relative to the rest of the Wholution, possibly until computing power increases by many orders of magnitude (e.g. perhaps "quantum computing" of the future).

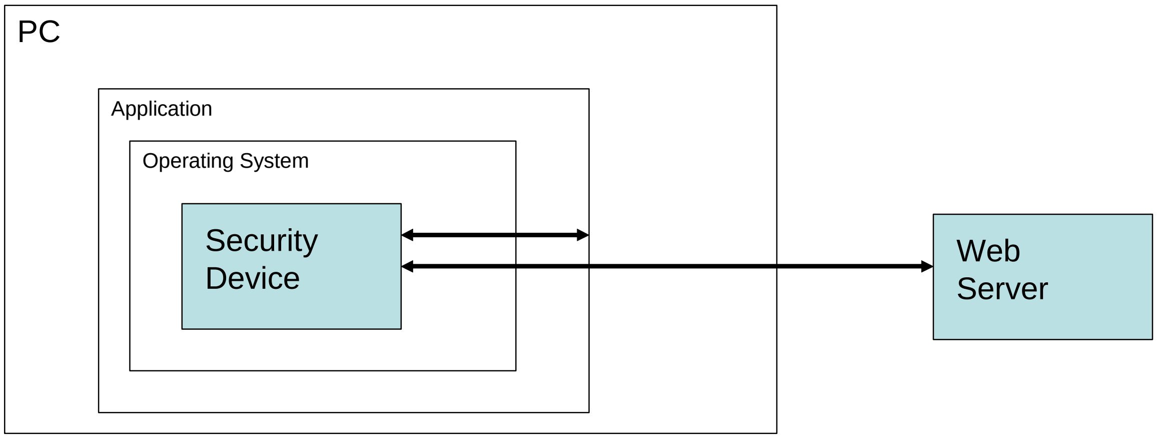
Examples of technology that include crypto API's are Microsoft CAPI, PKCS #11, GSC-IS/CACs and Javacard. Many crypto API's also include standard interfaces so any software can use them without requiring programming. (e.g. VPN technology can use CAPI or PKCS 11 to access a certificate/private key which is stored anywhere – on the disk, on a smart card or on a USB token).

Security Devices (Cryptographic processors)

Various hardware technologies are available to provide cryptographic services, store private keys (the private key is never exposed to the outside world), and provide enhanced speed for cryptographic operations (accelerators or co-processors – typically used for embedded or small portable devices). These include:

- Smart Cards and derivatives (SIM cards, USB tokens)
- Customized chips
- Co-processors

Secure Channels with a Security Device



If a hardware device is being used (e.g. a USB token, smart card, or TPM), the data that is transferred to and from the device should be secure. There are generally two types of systems that will connect to a hardware device: those that are on the host (e.g. PC), and those that are remote (e.g. over the Internet).

It is important for a remote device to have security as this data is crossing the Internet. Although it is possible to use another secure channel technology, like a VPN (aka. a tunnel), it's much more secure to have the encryption work endpoint-to-endpoint (i.e., between a server and the hardware device), thus no data (not even anything on the host) can compromise the data.

It is a good idea also for the data between a host (PC) application and the hardware device to also be encrypted. In today's environment, the PC application is not secure. Once any data is released from the hardware device to the application or OS, it can be compromised by hackers and viruses. However bus traffic tends to be even less secure, thus it is a good idea to encrypt that connection as well, despite the vulnerability of the application.

One way to setup a secure channel is similar to SSL (See Authentication and Key Exchange). Quick summary:

- Certificates are exchanged, validated, and a challenge is sent with a digital signature which verifies the holder of the certificate has the private key, and the certificate has been issued to a valid member of a "group" (e.g. issued by a particular vendor who is trusted).
- A symmetrical session key is generated, and the public key of the other party is used to exchange the session key
- The symmetrical key is then used to transfer data. A digital signature may be included to ensure data has not been modified in transmission. Salting may also be used to ensure data cannot be replayed.

A secure channel can also be set up with symmetrical encryption. However, the key management process is more complex (and thus more error prone), as keys need to be distributed securely.

Note that the use of secure channels assumes that the other entity (i.e., web server or application/OS) has a private key. In the case of the web server, the private key can easily be secure as web servers are typically secure. However, in the case of the application or OS, the private key must be embedded inside the software. This is an area of vulnerability. (The private key should not be stored inside the hardware device, because this would be a circular design.) However, providing secure channels still makes the system more secure because it is more difficult to hack into the application than into the bus.

Data and key storage

When using a secure hardware device, critical keys can be stored in the device. However, hardware devices typically have a small amount of memory. Thus, rather than storing all keys in the device, these keys (master keys) are typically used to encrypt other keys which are then stored on the system's hard disk, encrypted. In order to access the master key, typically a PIN or shared secret is sent to the hardware device to prove authorization to access the key. However, this presents several difficulties:

- If the PIN or shared secret is not encrypted it could be compromised. A secure channel could be used, but the value of a secure channel in protecting this data depends on several factors.
  - If the PIN is coming from a user, it doesn't have to be stored in the application, thus it is less likely to be discovered by hackers. However, this presents another problem which is what if there is a keyboard sniffer virus. Note that the probability of this causing a problem is small, but it is not 0%. It depends on the application. (Note that PINs are typically short, vs. keys which are much longer. A hardware device may be designed to prevent "dictionary attacks". So for example, if you try to guess the PIN, after a certain number of failed attempts, the system will lock up).
  - If the PIN or shared secret is not coming from a user (e.g. the application itself is retrieving data), then the shared secret must be embedded inside the application. This PIN could be encrypted, but then the key which encrypted it has to also be

stored in the application. One can encrypt keys upon keys, and hope to confuse a hacker, but there is still a reasonable chance that the system could be hacked, because in the end, there is always the ultimate master key which is vulnerable. (The ideal system is one in which even the designer can't hack it).

- If a secure channel is used, the same problem exists, the private key used to authenticate to the hardware device is inside software, which is hackable.

In addition, even if the PIN is protected and a secure channel provides additional protection, the decrypted data coming from the security hardware device, may be released into the application, which is unsecure.

What this all means is that although using a hardware module will increase the security of an application, the achieved level of security varies based on the objectives:

- The best protection is against theft of the system. If the system is off, and all data is encrypted, there is no chached data or keys floating around to steal. Note that the newest version of Windows (Vista) scheduled for 2006 has this build in with a feature called Full Volume Encryption (all data on the disk is automatically encrypted by the OS)
- For best security, the user will always have to type in a PIN, as master access keys then don't have to be stored inside the application. (You would want this anyway, i.e,. not to rely on OS login).
- If you're trying to protect against viruses (i.e., threats when the system is running and the user is using the system), protection is more difficult to achieve and a more careful assessment is required.

Application licensing and configuration

Similar to data storage, licensing and the need to control information can benefit from hardware security. However, similar problems exist – it's not easy to store all the data in the hardware device. For example, configuration data (the number of times a video can be played), can be encrypted and stored on the hard disk, where the key is stored in the hardware module. However, this requires the application to encrypt and decrypt the data, thus you must trust the integrity of the application. However, trusting the integrity of the application must go beyond just encrypting and decrypting configuration data. Although the best designed security is one where the design cannot hack the system, it's difficult to achieve in an untrusted environment (e.g., where you don't trust the user) because you can't rely on the user to not run pirated software. Some of this functionality can be implemented inside a secure hardware device, however, this has challenges:

- Limitations of memory.
- Requires security firmware running on the hardware device which is specific to the application.
- This firmware has to be distributed (and thus may be limited to special applications).
- There is still a design challenge in ensuring security as, ultimately, the application software that uses this secure hardware device is unsecure.

Securing system resources with the Operating System

Some Wholutions require that resources in the operating system are secure and/or controlled. For example:

- Access to network
- Access to exportable media such as flash drives, CD ROM drivers, printers
- All data must be encrypted inside OS and/or to hard drive

For example, the IT administrator of your company might not allow you to connect to the corporate network if another network connection is open.

This area is still evolving. For example there are various LINUX projects to produce such an environment.

Key configuration and Delegated management

Any secure hardware device is typically released with at least one key (either symmetric or certificate). In some cases this key can be used to configure the device remotely (e,.g. adding/changing software, licensing, other keys). However, there is typically an management issue around ownership of the key. Once a key is on the system, it can be used to add other keys and enable other applications However, these applications could be provided by multiple vendors who would need to be coordinated if the application requires anything more advanced than basic key and storage, or proof of identity (commands which could be provided by a generic application running on a smart card or TPM). Secure devices can use delegated management (i.e., issued by one vendor, but used by and securely controlled by one or more other vendors). But in order to allow multiple vendors to share a general purpose cryptographic platform, there must be an agreement regarding how delegated keys are used, who is trusted to issue the master key, and what platform design will be used.

Authentication and approval of transactions

By far, basic authentication is the most secure because it is a simple transaction between a server and the security device. However, as in the VPN scenario, there are other risks related to the use of the system *after* authentication has occurred. Furthermore, authentication typically requires one to type a password and/or scan their fingerprint. It is important for some users to ensure that passwords and fingerprint scans cannot be sniffed by a virus (e.g. a hacker couldn’t not steal your password and login to your account from their system, but they could possibly take over your system and log into your account).

For the approval of transactions (e.g. credit card transactions over the web or at a point of sale), the need to ensure that a display showing the transaction before it is approved by the user is secure. It is also possible to use a secure keyboard to ensure that a virus cannot sniff and/or playback your keyboard strokes. However, these more advanced features require an integration between the hardware and the operating system or software using that hardware.

Using a secrity device as a cryptographic processor

A device can be used as a cryptographic processor in several ways:

- Just as a accelerator (i.e., key storage and key security is not a critical issue because the keys are freely located in the unsecured application anyway)
- As a mechanism to protect private keys for identification (the most robust application)

If the device is only being used as a cryptographic process, the security design is much simpler.