View on GitHub

OSVTAC Recommendations

San Francisco Open Source Voting System Technical Advisory Committee (OSVTAC) Recommendations

Open Source Voting System Project Recommendations

(Approved by OSVTAC on January 18, 2018.)

Last posted: April 12, 2018

- Introduction & Table of Contents (for multi-page version)
- Single-page version (long, can be used for printing)

8. Appendix

[Section added by Carl Hage: for discussion at March 8, 2018 meeting.]

8.1. Appendix 1 - Data and Software Security

This section contains a brief description of cryptographic technology and how it might be used within a voting system. The goals include:

- Proving data is unaltered from the original (by error or intent)
- Proving data is created on a particular machine and certain time-frame
- Insuring no data in a collection has been inserted or removed
- Authenticating input data used by software
- Linking a set of output data with specific versions of input data and software
- Establishing a chain of custody for data and software
- Insuring software running on a machine is unaltered and certified

- Insuring upgrades are only authorized and certified

8.1.1. Cryptographic hash a.k.a. *digital fingerprint*

The basis of modern data security is the *cryptographic hash*, sometimes just called a hash. A cryptographic hash is also sometimes called a digital fingerprint because it is a code that can uniquely identify a piece of data, analogous to how a human fingerprint can identify a person.

The hash is a number with many digits (usually measured in bits, e.g. 256, equivalent to about 77 decimal digits) created from an arbitrary size set of data. Any change in the input data will almost certainly have a different hash value. The large number is typically represented as a set of hexadecimal digits (0-9, a-f), sometimes with a hyphen inserted for readability, or using upper and lower case letters and numbers.

When downloading or transferring data, it is customary to compare the hash derived from the data download with a known original hash. This is primarily for detecting errors.

Another common use of a cryptographic hash is to identify a particular revision of a file. The hash identifies a particular set of content, independent of file modification times. Each revision can be tracked by it's hash code, and future edits that revert back to some same prior content will have the same hash value.

A cryptographic hash also has the property that it is "one way", i.e. you can convert input data into a hash code, but cannot convert the hash code back to the original data. Also, you would not be able to invent some altered piece of data with the same hash result. [Of course if the data hashed is an English word, hashing all words in a dictionary would reveal the original.]

There are several hash algorithms available, but the most common currently in use now is SHA-2 (Secure Hash Algorithm version 2). The 512 bit version is called SHA-512, currently used by the San Francisco elections department for downloadable election results files.

In summary, a cryptographic hash can:

- Insure data is unaltered from the original used to create the hash
- Identify a unique set of data content

8.1.2. Encryption with public key cryptography

With public key cryptography, some data may be encrypted (cloaked) and sent to sent to a recipient, and only the intended recipient can decrypt that data. The key to encrypt and key to decrypt are separate. The recipient publishes the encryption key (the *public key*), used by the sender, then after receiving the encrypted data, uses a second decryption key (the *private key*) a secret known only to

that recipient.

Public key encryption is used to send confidential data without the possibility of being intercepted and read. In the context of voting, early voting data could be sequestered by encrypting it with a public key, and stored until election day, when the private key is used to unlock the sequestered data.

Public key cryptography is also used by https which could be used to secure voter privacy in the communication network used for electronic poll books.

Public key encryption could also be used to make the electronic equivalent of the paper vote-by-mail or provisional envelope that hides the paper ballot inside. If an electronic transmission of a ballot is required by a military or overseas voter, the ballot information could be encrypted with a public key. Later, a batch of these encrypted electronic ballots would be sent to a special software system, where voter identification is removed, the ballot order randomized, and the decrypted ballots are written out. The private key would be held in escrow and only passed to the software system used to strip voter identity and extract the ballot data. Once ballots have been extracted, the private key can be destroyed to preserve voter identity in archived data. An electronic version of remote accessible vote-by-mail probably would need to layers of encrypted wrapping– one for the ballot itself (without voter identification), and a second level that includes private voter identification information, encrypted so only the election administration can read it and insuring the message content could not be altered even when sent via potentially insecure email. [After receipt of the encrypted outer layer, the election admin could send the voter a receipt, which might include the hash of the encrypted ballot data received.]

In summary, public key encryption can:

- Hide data so only the recipient can read it
- Insure data is unaltered (if used with a cryptographic hash)
- Sequester data if the private key is sequestered
- Separate associated voter identity from electronic ballots, except within the software system that extracts ballots and detaches voter identification.

8.1.3. Symmetric (shared secret) key cryptography

An alternative to public key encryption is a single *symmetric key*, shared between sender and recipient, rather than the separate public/private key pair. This was used prior to the development of public key cryptography, but is still used when transmitting or encrypting large amounts of data since symmetric keys are faster, and public key cryptography does not work well with large amounts of data.

For large data encrypted with public key cryptography, a unique random symmetric key is created by the sender, then that key is encrypted using the public key, included in the data sent. The recipient

uses the private key to obtain the symmetric key, then used to decrypt the data. The secured https protocol uses this method for security.

Except for the single-use symmetric keys used with public key encryption, symmetric shared secret keys are considered insecure and should be avoided in voting systems.

8.1.4. Digital signatures

A digital signature is a specialized form of public key cryptography, except the role of public key and private keys are reversed– the secret private key is used to encrypt data, and anyone can use the public key to decrypt data. A *digital signature* is an encrypted form of the cryptographic hash of the signed data. Since the private key is used for this encryption, only the owner of that key can create the encrypted hash. The public key can then be used to decrypt the hash value, and compare that against the recreated hash of the signed data.

In summary, digital signatures can:

- Insure the owner of the private key created the signature
- Insure data is unaltered from the signed original

8.1.5. Specialized hardware to hold private keys - Hardware Security Modules (HSM)

Often public key encryption and digital signatures use a private (secret) key stored in a computer file, then used by software to decrypt or sign data. Secrecy of the key relies on limiting access to that file (or computer memory). If that computer is compromized, then the private key could be stolen and used to forge new signatures or decrypt protected data.

To solve this problem, specialized hardware is available to generate and store private keys within a microchip. The stored key can be used to decrypt or sign data inside the chip, but otherwise the secret key never leaves the chip. Sometimes these microchips are placed in a USB stick or smart card– when the device is inserted into a computer, then the computer can use the device. Typically, a password is required to enable the device. These devices are typically known as a *Hardware Security Module* (HSM) but sometimes called a *hardware token* or *security token*.

Hardware devices can be quite sophisticated in protecting the secret key. New devices now protect against monitoring the electric power consumption used to perform the cryptographic calculations– the power required otherwise giving clues to the key values. Some chips include special shielding within the manufactured chip (preventing analysis with an electron microscope), or are built such that any monitoring inherently changes the key value.

Most hardware devices have a tamper-detect signal, which can erase the stored keys. The encryption device or even a small microcomputer that can run data security applications can be enclosed within a

wire grid case connected to the tamper-detect. Any attempt to drill into or open the case breaks a wire and deactivates the device.

Some hardware encryption devices include a serial number for each digital signature created. A log of all signature data, one for each serial number, can be used to insure that this device could not have been used to create any additional signatures outside of the log, i.e. the device could not have been temporarily stolen and used to create unauthorized signatures. If the log containing all prior signatures (1 to N) is signed (creating signature serial N+1), then that proves no other signatures have been created prior to the latest N+1 signature.

In summary, hardware encryption devices can:

- Prevent a private key from being revealed, even to it's owner
- Insure only that specific device was used to create a signature
- Protect private keys from being revealed, even if the device is stolen (depending on tamper protection)
- Insure that no signatures have been made with that key other than logged signatures when used with a hardware device that signs data including an incremented serial number

8.1.6. Trusted Platform Module (TPM)

TPM is an international standard (ISO/IEC 11889) for a secure cryptoprocessor that is built into many recent computers. (Motherboards have a connector to install the TPM.) The TPM standard includes a number of functions, including an ecryption-decryption-signature component (functioning as a HSM), and a means to implement secure boot. Each TPM chip has a built-in unique identity set by the manufacturer.

Some TPM implementations are based on software or firmware (BIOS EPROM). Within a virtual machine or container, the host hypervisor can provide a virtual TPM.

Drivers (including open-source) are available to access the TPM hardware within standard cryptographic software to perform the secure operations.

A TPM generally does not have the same level of tamper resistance as most HSM implementations– settings can be changed by software (or malware) with root-level (physical or sysadmin) access. However, cryptographic security is much better than software/file stored keys.

Hardware procurement for general purpose computers should probably include a requirement to support hardware TPM.

8.1.7. Securing a data sequence with block chains

If a new block of digitally signed data (could be a ballot image or batch of cast vote records) includes the cryptographic hash of the prior block in a sequence, then the latest block also validates the prior block of data. The prior cryptographic hash in each block forms a chain of hash values, validating all blocks in the chain, and also insuring no blocks have been inserted or removed. Since each new signed block validates the whole chain of prior blocks, each added signature validates the chain even if a prior signature key is compromised.

Note if a block of data is a ballot image, then the block chain is inherently ordered data, so file order cannot be randomized. To protect voter privacy, the order of voters casting ballots linked into a chain must not be recorded. If CVR order must be randomized, then it might be possible to collect a batch of ballots (say a block of 100 CVRs), randomize the CVRs within the block, then write out a new signed data block.

For historical data, e.g. where each block is the definition and official results of an election, then a block chain of election results over the years revalidates the history. If new and better encryption devices are developed in the future, then the latest technology validates all data.

Block chains could also document a set of revisions of released data– each new version could include the hash of the prior version. The collection of data revisions becomes a block chain.

The signed block chain is used for the public ledgers of cryptocurrency transactions (e.g. bitcoin). Each new signed block of transactions validates all past ledger data.

In summary, block chains can:

- Insure there is no missing or inserted data in a collection
- Each addition to a collection revalidates all prior data
- Enables new encryption technology to validate a historical archive, insuring no data has been altered, inserted, or removed.

8.1.8. Certificate authorities

The sender of data to be encrypted or the recipient of signed data needs to know if the public key received matches the intended recipient of encrypted data or is the authentic owner of the signature keys. The public key could be obtained directly from the owner of the keys, but that exchange can be complicated and it could be difficult to prevent the stored public key from being altered.

To solve this problem, a *certificate* is created to authenticate a public key. At the lowest level, the certificate contains the public key signed by the owner of the private key (so the public key is valid for itself, and works with a private key used by it's owner). A trusted third party, known as a *certificate authority* (CA), is used to sign the certificate after it receives the original certificate and authenticates the owner by some means. The signatures of the certificate authority itself must be validated, so the CA in turn has a certificate validated by a higher authority up to the top level known

as the root.

To implement https secure web communications and to support encrypted and signed internet email, browser and email software manufacturers have an agreed upon set of *root certificate authority* organizations. These root CAs in turn may authorize a second-level CA, and manages it's own keys stored in hardware devices held within secure data centers. While CAs generally have high security cryptographic hardware, the typical internet servers or email applications rely on software-stored private keys.

From wikipedia, "A *public key infrastructure* (PKI) is a set of roles, policies, and procedures needed to create, manage, distribute, use, store, and revoke digital certificates and manage public-key encryption." The terms PKI and CA are sometimes used interchangeably, but refer to the whole process and organization of issuing certificates.

For secure voting applications, the usual chain of certificate authorities (PKI) used for internet applications might not be desirable. Instead, the voting equipment, and software to authenticate voting data might be configured to accept certificates issued from a special source, e.g. a certificate authority operated by the Secretary of State. Certificates would only be issued to election officials within that state, and only for keys used with certified voting equipment.

If historical data is saved with a block chain of cryptographic hashes, data could be secured initially with the usual certificate authorities used for the Internet, then later signed with certificates issued from some future state or federal run CA, specific to secure voting data.

In summary, a certificate authority:

- Certifies that a public key belongs to a named owner
- Certifies that some procedures have been done to authenticate owners
- Certifies that owners of private keys comply with security requirements
- Are trusted by users of signature verification or encryption

8.1.9. Digital notaries

A trusted third party (e.g. a Secretary of State) could provide a service to co-sign data received and archived, providing a similar role as a notary public with paper documents. The digital notary could accept data transmitted in an agreed upon way, and require signatures from selected authorized certificates. A time stamp included with the notarization signature could also validate the time and sequence of notarizations. A notary digital signature also authenticates data, independent of the original signatures, insuring it is unaltered and complete.

Another aspect of a notary provided by government officials is that notarized data can implicitly marked as an official filing of a public record. A county elections official might also create added digital signatures analogous to a notary to identify official certified data. Output data files created by

voting software would be digitally signed by the machine running the software, possibly using a software-stored private key. When data is final and certified, the added signature by the elections official adds an endorsement of the original signed data.

To trust time stamps from a digital notary, the notary logs must be signed and notarized by other third parties and/or checked by the public at large.

In summary, a digital notary can:

- Record the time of filing, validating a timestamp in an original signature
- Validate that data is official
- Provides a trusted single signature to authenticate data
- Provide a signed log with a block chain of all notary transactions with a link to notarized data.
- Indicate an endorsement of official certified data

8.1.10. Cryptographic hashing of daily public data

A digital notary service can prove data has been signed no later than the date of notarization, but by itself the time in the original signature cannot be proved to be correct. One possible solution would be to include a cryptographic hash of semi-random public data updated daily, e.g. the top 1000 NASDAQ stock closing price and number of shares traded. Anyone could download the historical data and recompute the hash for that day. The inherent volatility of stock price and volume leads to an unpredictable random number hash, but available as a historical record.

Another possible source of public hash would be the latest bitcoin block chain hash. [Bitcoin hashes get created about once every 10 minutes.]

If digitally signed data includes the hash of such current public data, then that proves the signature could come no earlier than when the data was first available.

The public data hash could also be used to create a seed for a random number generator used for determining candidate order or picking ballots in a Risk Limiting Audit. [This would be an alternative or addition to a public roll of dice.] As long as the time for choosing the hash was set in advance, the subsequent value would be effectively random.

In summary, a public data hash can:

- Prove signed data was created no earlier than the public data
- Create a random number without relying on trust of machines or procedures

8.1.11. Temporary private keys

Data security for election day data recording (e.g. by a precinct scanner) can be enhanced by using a

temporary public/private key created at the start of the election, then the private key would be erased at the close of the election. A voting machine would have a permanent private key held in hardware that uniquely identifies signatures from that device. It could create the temporary key and publish the private key with a certificate signed by the permanent hardware key inside that device.

During the election, data can be recorded in a signed block chain by both keys. [A block chain could be used to prevent loss of signatures in case of a shutdown, vs signing the hash of all data at the end of the day.]

At the close of the election, the machine destroys the private key, but the public key used to validate signatures still exists. The permanent key proves data originated on that particular machine, and destruction of the temporary key prevents that machine from being used to later sign fraudulent data. Poll workers retaining an end-of-day printout can validate the integrity of the hash for an archive of that precinct data. [Of course, the cast paper ballots are the true means to insure data integrity.]

Another possible use of a temporary key might be to separate provisional or remote ballots sent electronically from voter identity recorded in submitting ballots. The provisional ballots (without voter identity) would be encrypted with a temporary key held in escrow. A software program would extract these encrypted ballots from voter identity, then pass the plain encrypted data to another software package that has access to the private key and can decrypt the data. After the decryption succeeds, the key is destroyed so stored logs of the original data can't be used to reveal the secret ballots.

Note, each encrypted ballot would also need to include a random number (added to the end of the data) that is also stripped during the decryption– otherwise the voters identity might be disclosed by re-encrypting the extracted ballots. The decryption program could randomly sort the extracted ballots and separately the extracted random numbers. Voters could verify receipt of their ballot using a log of the cryptographic hash of extracted ballots prior to stripping the random number added. [However, this would not be end-end verification since the extraction software could, in-theory, alter ballot data.]

In summary, temporary private keys can:

- Prevent a machine from signing data outside the life of the key
- Limit decryption of data to a specific point in time

8.1.12. Securing the chain of custody for data

Besides using digital signatures on output data files, protecting their integrity, log files produced by application software could include the cryptographic hash of all input files, the hash for intermediate files (files derived from input data, used to create output data, but not stored as output data), and hash values for output files created. In addition, if hash values are available for the application software, operating system, or container, these could be included in a log. All these tie the output data with

specific versions of the input data and software. The validation of the whole software and data chain serves both a security and auditing role, as well as an error detection and analysis role. (A error in a file used for input can be associated with the whole chain of output files via hash values in log files.)

Output data can be digitally signed and linked with a digitally signed log file. (Actually, if a log file contains the hash values for output data, signing the log indirectly signs the output data. However, a generic signature validator might not be able to read application-specific logs.)

Data changes and matching data content can be tracked by the hash values– if the hash of an input or output file changes, then something is different, or if the hash is the same, data is unchanged, even if recreated. An auditing tool could scan logs and identify an output file created with obsolete input data. In that case, the output data could be recreated– if the output hash values remain the same, then the changed input had no effect.

Security sensitive applications, e.g. a vote totaler, must check the cryptographic hash values of all input data read. But ideally all applications could collect and validate input data. Recording the hash values in a log is all that is needed rather than full signature checks, since the full checks of input and output signatures can be performed later as a separate process/check.

With open input data, output data, and open software, the public at large can check input data, recreate output data, and validate content. Digital signatures preserve the integrity of the original data and identify and authenticate official filings.

In summary, using signed log files with input/output/software hashes can:

- Associate specific versions of input data with output data
- Track versions of software used to create output data
- Identify altered or changed input data in the chain of software processing
- Identify reprocessing required after making an error correction or update
- Insure certified input data and software is used to create certified output data
- Validate the integrity of a collection of data across multiple processing steps
- Enable public auditing with open data and software

8.1.13. Secure boot and software integrity

Digital signatures attached to input and output data can validate the integrity of data, but there is also a need to secure software running on voting equipment. Checking digital signatures of the software running on a machine can be used to determine if software has been altered, and also limit upgrades or installation of new software to be only certified versions.

Note, no computer can be completely secure, since sophisticated attacks like fake disk hardware, for example, could deliver real data during a signature check, then fake data later. Thus voter verified paper ballots and audits provide the true security. However *secure boot* and validating digitally signed

software during the machine startup can provide a high degree of integrity, especially compared to traditional computer systems used for voting.

Securing software requires validating the integrity of several layers of software– the firmware stored in the microchips, the operating system and generic software, and the voting application software.

8.1.13.1. Secure firmware

The lowest level software is *firmware*, software stored inside a microchip during the manufacturing process. Most microcomputers have a way for electronic equipment to look inside the chip for testing, debugging, and to load firmware. A secure microcomputer would need to permanently disable this capability or have a mode where all prior memory in the microchip is erased when enabling debug access.

A *secure boot* relies on a small piece of firmware in the microchip that cannot be altered, and this firmware runs when the microchip is powered on. The purpose of the secure boot firmware is to scan separate more sophisticated boot loader firmware that can load or validate the operating system and possibly application software. The secure boot firmware scans the next stage boot loader and checks for a valid digital signature. If the signature does not match, the machine stops. Secure boot firmware can also be used to make a change or upgrade to the second level boot firmware, e.g. by reading new software as input data, then validating a signature.

Ideally the lowest level secure boot firmware would require a specific public key from a certification authority of voting equipment, not the key of a general equipment manufacturer, e.g. Microsoft (used to securely boot Windows). Also, ideally the key required would be permanent and unable to be changed (the microchip would need to be replaced).

If possible, the secure boot firmware would be readable by ordinary software but not alterable (except possibly for certain secret data like a private key). Thus external software can read and validate the secure boot firmware as well.

Not all microcomputers have this capability of unalterable firmware stored within the microchip. The “secure boot” from some systems might still allow special electronics to change the signature validation key or firmware. Complex microchips (e.g. Intel Pentium) rely on an external microchip to store firmware.

When selecting equipment for voting applications, the secure boot firmware capabilities should be one factor considered. Using COTS equipment might not provide all ideal capabilities.

8.1.13.2. Secure operating systems

A typical secure boot would use firmware to validate a more complex and upgradable second stage

boot loader, which might in turn validate and start a third stage boot loader that could scan all operating system software and validate a digital signature, prior to loading the operating system. These second or third stage boot loaders could be configured to require a signature from an authority that certifies voting equipment. This way, the machines could only run certified software.

Any upgrades to the operating system would also require validation before changes are made, and re-validating the complete set of software files (e.g. new secure boot).

Besides software, the parts of the Operating System that need to be validated include configuration file data (system settings). Some operating systems might not store configuration data in a way where unchangeable data can be isolated and validated.

Ideally boot loaders and operating system software would be stored in read-only memory, writable only during an upgrade process with a physical switch set.

The ability for computers to validate operating system software that runs on the machine should be part of the evaluation and selection process for computer equipment. Validation includes updates.

8.1.13.3. Secure application software

The validation of voting application software could be separate from validation of the operating system, or could be integrated with OS validation. An operating system might re-validate application software each time the software starts. Most certainly, each version of sensitive voting software should be certified by a recognized authority (e.g. Secretary of State), and signatures from that authority checked.

8.1.13.4. Securing applications with containers and virtualization

In some cases additional security can be obtained by running application software within a *virtual machine* (VM) or *container*, a form of virtualization that uses part of the host operating system. From the perspective of the application software, it appears to be running on a completely separate computer, isolated from applications running in other virtual machines or containers. Rather than securing a large set of operating system and application software running on a computer, a smaller less complex host operating system (known as a *hypervisor*) provides controlled isolated access to applications.

A container is a relatively new form of virtualization where a combination of operating system software and configuration files can be combined with a pre-configured set of application software. The containers do not need to include all of the operating system, as they can use the base level of a host operating system, but otherwise “contain” the equivalent of a pre-configured virtual computer. Each application can run in a different container, so a compromize in one application would be unlikely to affect an application in another. Each time the application is started, the container can be

validated with a digital signature check prior to booting the container and starting the application. When the application terminates, it is equivalent to shutting off a machine, and the next time the application starts, it reverts with the same validated container file.

Besides securing applications, containers are a convenient way to package and distribute complex applications. A secretary of state that certifies voting systems needs to have a complete package of operating system, applications, and configuration files. A container is created by layering pre-built operating system and COTS software (e.g. software libraries, database servers, etc.) containers with installed application software. Each of the individual container layers and combined final container can be updated, distributed, and validated independently, including validation by the public at large if made with open source software.

Note, storing sensitive data like passwords or private keys within a container or virtual machine could be a problem, since a compromised host operating system could peek inside the VM or container and extract that sensitive data. Thus private keys are best maintained in external secure hardware, though a compromised host might also has access to that hardware.

Some microcomputers have a complex layering of security levels, the secure boot and cryptography at the lowest level, a special hypervisor level that can control multiple operating systems, each operating system controlling isolated applications.

8.1.14. Signed digital data formats

There are several standards in common use for formatting digital signatures, and several ways of associating the signatures with the signed data. While selecting a single format to create and check input and output file signatures will be important, it might be useful to insert additional application-specific signatures, e.g. on PDF files, so signatures appear in Adobe Acrobat Reader, or in the git gpg format for github posted collections.

8.1.14.1. Isolated signature files

One method of associating a signature or just the cryptographic hash with a data file is to write the hash code or signature in a separate file, e.g. one with the same name but a `.dsa` or `.sha` file extension.

8.1.14.2. Application specific signature insertion

Several application specific file formats have standards for attaching digital signature data. These are recognized by

Application-specific formats include:

- PDF

- XML (XML-DSig)
- JSON (JWS)
- Internet mail message content (S/MIME)
- Software installation package formats (e.g. RPM, DEB)

Generally, these formats only work with the application specific data, though S/MIME can be used to wrap any text or encoded binary file.

8.1.14.3. JAR files

One format for signed data widely used in the Java programming and Android community is the JAR file. This is an extension of the usual `.zip` compressed archive file (called "archive" in MS-Windows), where some added special files are inserted into the `.zip` archive, in a directory path called `META-INF`. The contents of a JAR file can be extracted with the usual `.zip` software.

The `META-INF/MANIFEST.MF` file is a "manifest" containing the names of all regular files (all except the `META-INF`) with the cryptographic hash of each. Digital signatures added are actually signatures of this manifest. All data files are then signed indirectly via the hash codes in the manifest.

Files in the `META-INF` directory ending in `.SF` and `.RSA` or `.DSA` are signatures, with a base file name corresponding to the signing party. Data can include multiple signatures, e.g. from a notary or distributor. The `.DSA` or `.RSA` are the actual signed data (in DSA or RSA format), which is a signature of the `.SF` file, containing the hash value of the manifest.

Note, data can be read directly from the JAR archive, or all files can be unzipped into regular files. Signatures can be validated either zipped or unzipped. [Signatures or just manifest hash values can be added to a working file tree by creating the `META-INF` subdirectory.].

Some characteristics of a JAR archive file include:

- A single signature can apply to a collection of data files
- Any file format can be represented, text, binary, or a file directory tree
- Multiple signatures (from multiple parties/systems) can be added
- Data is stored efficiently in compressed form
- Is usable with ordinary zip software, no cryptographic software required
- A zip file is convenient for download, distribution, and archival
- Zip software can convert differing text file line ends between Windows and Mac/Unix when packing/unpacking
- Data integrity across large collections of files can be managed efficiently by scanning the hash codes in the manifest files. (E.g. easy to track changes in updated versions or extracts by comparing the hash values.)

8.1.14.4. sha256sum files

The core utilities as part of a standard linux distribution include the `sha256sum` command, that can scan a set of files to compute the SHA256 cryptographic hash and produce an output file format listing the file names and hash values. This command can also read the hash summary file to validate a collection of same-named files. A digital signature on the sha256sum file can be used to effectively sign the whole collection of files listed.

Rather than use a zip file extended with digital signatures using the JAR format, a custom means to sign and validate collections of files could be created with some scripts using the `sha256sum` command with standard digital signature software (e.g. `openssl` or `gpg`).

8.1.15. Certification of cryptographic hardware and software

The National Institute of Standards and Technology (NIST) certifies cryptographic hardware and software (FIPS 140-2). Federal Voluntary Voting System Guidelines (VVSG) require use of FIPS 140-2 certified cryptography.

The FIPS 140-2 has several levels of security certifications, 1 at the software integrity level, 2 with tamper resistance, level 3 protection even with physical access, and level 4 the highest level with protection against physical penetration.

Fortunately, the open-source OpenSSL package widely in use is certified, and cryptographic chip manufacturers make an effort to get thier products and software libraries (usually based on OpenSSL) certified.

Some TPM chips have FIPS 140-2 certification with level 2 protection.

OSVTAC Recommendations maintained by carl3

Published with GitHub Pages