Spot-On

Echo Communications Software

Introduction....3
Accounts....4
Adaptive Echo....5
Block Cipher Modes of Operation....6
Capabilities Sharing....7
Cascading Encryption....8
Communication Methods....9
Communication Model....10
Communication Sessions....11
Compilation Process....12
Configuration Settings....13
Constant-Time Comparison Function....14
DTLS....15
Echo....16
Echo Public Key Sharing....17
Electronic Mail....18
Electronic Mail Forward Secrecy....19
Encrypted and Authenticated Containers....20
File Encryption....21
Fragmented StarBeams....22
Key Derivation....23
Hybrid System....24
Local Private Application Interfaces....25
McEliece....26
Multicasting....28
Multiple Devices....29
Non-Volatile Congestion Control Memory....30
NTL....31
Pass-through Devices....32
Poptastic....33
Problems....34
Public Key Infrastructure....35
Public Key Sharing....36
Secure Memory....37
Sessions....38
Socialist Millionaire Protocol....39
Sources of Entropy....40
Surreptitious Forwarding....41
Two-Way Calling....42
Verifying Ownership of Public Keys....43
Wide Lanes....45
References....46

Introduction

Spot-On is a dolphin-propelled science project. The software is composed of two separate applications, a multitasking kernel and a user interface. The two applications are written in C++ and require the Qt framework as well as an assortment of libraries. Qt versions 4.8.x and Qt 5.x are supported. Spot-On is available on FreeBSD, Linux, OS X, OS/2, and Windows. In addition to supporting the x86 and x86-64 architectures, ARM, PowerPC, and SPARC are supported without any special provisions.

Please note that the Echo algorithm and its name are not derived from Ernest J. H. Chang's 1982 Echo Algorithms: Depth Parallel Operations on General Graphs paper.

Software source is available at https://github.com/textbrowser/spot-on.

Accounts

Spot-On implements a plain, and perhaps original, two-pass mutual authentication protocol. The implementation is well-defined with or without SSL/TLS. Synchronized clocks are required. The protocol is weakened if trusted SSL/TLS is neglected, however. Please see the paragraph at the end of this section for additional details regarding the weakness. The Accounts procedure is as follows:

1. Binding endpoints are responsible for defining account information. During the account-creation process, an account may be designated for one-time use. Account names and account passwords each require at least 32 bytes of data.
2. After a network connection is established, a binding endpoint notifies the peer with an authentication request. The binding endpoint will terminate the connection if the peer has not identified itself within a fifteen-second window.
3. After receiving the authentication request, the peer responds to the binding endpoint. The peer submits the following information: $H_{\text{Hash Key}}(\text{Salt} \| \text{Time}) \| \text{Salt}$, where the Hash Key is a concatenation of the account name and the account password. The SHA-512 hash algorithm is presently used to generate the hash output. The Time variable has a resolution of minutes. The peer retains the salt value. Please note that the peer may submit credentials without having received an authentication request. This is acceptable.
4. The binding endpoint receives the peer's information. Subsequently, it computes $H_{\text{Hash Key}}(\text{Salt} \| \text{Time})$ for all of the accounts that it possesses. If it does not discover an account, it increments Time by one minute and performs an additional search. If an account is discovered, the binding endpoint creates a message similar to the message created by the peer in the previous step and submits the information to the peer: $H_{\text{Hash Key}}(\text{Salt} \| \text{Salt}_3 \| \text{Time}) \| \text{Salt}$. The authenticated information is recorded. After a period of approximately 120 seconds, the information is discarded.
5. The peer receives the binding endpoint's information and performs a similar validation process, including the analysis of the binding endpoint's salt. The two salt values must be distinct. The peer will terminate the connection if the binding endpoint has not identified itself within a fifteen-second window. Peers will ignore unsolicited authentication requests.

Please note that the Accounts system may be promoted by including an encryption key. The additional key will allow for finer time resolutions.

If SSL/TLS is not available, the protocol may be exploited. A relay station may record the values in the 3rd step and subsequently provide the information to the binding endpoint. The binding endpoint will therefore trust the foreign connection. The recording device may then seize the binding endpoint's response, the values in the 4th step, and provide the information to the peer. If the information is accurate, the peer will accept the binding endpoint's response. Therefore, the Accounts protocol is susceptible to impersonation.

The use of nonces and timestamps, in conjunction with congestion control and the retaining of distributed nonces, should prevent re-ordering, reflection, and replay attacks.

Adaptive Echo

The Adaptive Echo is a complement to the Echo and substantiates the opinion that the Echo is a malleable method. Endpoints that bind multiple parties may optionally define Adaptive Echo tokens. Adaptive Echo tokens are composed of authentication and encryption keys as well as details about the choice algorithms. If configured, binding endpoints are able to permit or restrict information travel based on the content of the data. As an example, peers that are cognizant of a specific Adaptive Echo token will receive data from other cognizant peers whereas traditional peers will not. Binding endpoints therefore selectively-echo data.

The Adaptive Echo behaves as follows:

1. A binding endpoint defines an Adaptive Echo token. The information must be distributed securely.
2. A networked peer having the given Adaptive Echo token generates $H_{\text{Hash Key}}(E_{\text{Encryption Key}}(\text{Message} \| \text{Time})) \| E_{\text{Encryption Key}}(\text{Message} \| \text{Time})$ where the Encryption Key and Hash Key are derived from the Adaptive Echo token. The generated information is then submitted to the binding endpoint as Message || Adaptive Echo Information.
3. The binding endpoint processes the received message to determine if the message is tagged with a known Adaptive Echo token. If the message is indeed tagged correctly, the Time value is inspected. If the Time value is within five seconds of the binding endpoint's local time, the message is considered correct and the peer's presence is recorded.
4. As the binding endpoint receives messages from other peers, it inspects the messages to determine if the messages have been tagged with Adaptive Echo tokens. This process creates a network of associated peers. Because peers themselves may be binding endpoints, the Adaptive Echo may be used to generate an artificial trust network.

Adaptive Echo is susceptible to eavesdropping. As an example, if a message that is tagged with an Adaptive Echo token should travel through one or more peers to reach a destination, the peers may record the message and subsequently replay the message to a binding peer. The replay must occur within the acceptance window of the message. Additionally, the binding endpoint's congestion control container must not already contain the message. If both conditions are met, the binding endpoint will consider the peer as trustworthy.

Block Cipher Modes of Operation

Spot-On uses CBC with CTS to provide confidentiality. The file encryption mechanism supports the GCM algorithm without the authenticity property that's provided by the algorithm. To provide authenticity, the application uses the encrypt-then-MAC (EtM) approach. The Encrypted and Authenticated Containers section provides more details.

Capabilities Sharing

Peers may exchange capabilities information. Currently, the information includes the UUID, the echo mode, and the preferred lane width. Please note that a server socket must adhere to its peer's settings. Although an administrator may modify the echo mode and the lane width on a particular socket (remote neighbor), the kernel will override the settings so as to conform to the client's preferences.

Cascading Encryption

Spot-On implements multiple encryption. The general implementation is as follows:

1. Retrieve random authentication and encryption keys.
2. If available, retrieve previously-established authentication and encryption keys. The credentials in #1 are not compared with the credentials in #2.
3. Generate a hybrid bundle: RSA(#1) || AES(message) || HMAC(RSA(#1 keys) || AES(message)).
4. Generate a bundle from the data in step #3 using the keys from step #1: AES(#3) || HMAC(AES(#3)).
5. If SSL/TLS is available, funnel the bundle from step #4 through the SSL/TLS layer.

Communication Methods

Spot-On supports Bluetooth, SCTP, TCP, and UDP (multicast and unicast) communication methods. Both IPv4 and IPv6 are totally supported. Some communications portions also support a variety of proxies. For TCP-based and UDP-based unicast communications, OpenSSL is supported. Spot-On distributes data with or without SSL/TLS. Please note that magnet distribution violates this principle and therefore requires SSL/TLS. In essence, the application is generally transport-neutral.

Please note that Bluetooth and SCTP support require operating system support.

Communication Model

Spot-On mostly assumes an asynchronous communication model. The Accounts, SMP, and Two-Way Calling systems require responses.

Communication Sessions

This section describes some common methods which may be employed against Spot-On in an attempt to destabilize the communications of two parties.

Various attack methods include: flood attack, re-ordering attack, reflection attack, and replay attack.

Public Spot-On devices are susceptible to flood attacks.

Save for Local Private Application Interfaces, Spot-On does not guarantee orderly delivery of data and is therefore susceptible to **re-ordering**. However, this attack method is not considered detrimental as Spot-On is not orderly. Some sub-systems such as Accounts and SMP should be resistant to this particular attack.

In a **reflection attack**, an adversary may capture a message sent from party A to party B and return the message to party A. For this attack to be successful in messages involving public keys, it must be possible for the originator of the message to be able to decipher its original message. Private-key communications are susceptible to reflection attacks, however, properly-defined congestion control and timestamps should aid against reflection attacks.

In a **replay attack,** an adversary may replay a valid message. Discounting congestion control, the message may appear as if it was sent recently. All private-key portions of Spot-On contain timestamps. Most public-key portions also contain timestamps, e-mail being an exception. Congestion control and timestamps should provide some resistance against replay attacks.

Computing and memory resources are required to prevent the aforementioned attacks.

Compilation Process

Spot-On builds are not digest-reproducible as the compilation process embeds the date and time of the compilation. In order to produce digest-reproducible builds, the SPOTON_DATELESS_COMPILATION option should be enabled during the generation of the project files. For example, qmake DEFINES+=SPOTON_DATELESS_COMPILATION -o Makefile spot-on.qt5.pro.

One should then expect results similar to the following.

date && sha512sum ./Spot-On ./Spot-On-Kernel

Fri Jan 6 10:27:21 2017

0fbcf8e8a82dacd9825c42cf88d2d5a70d87965da20a3bfc5f87b8f634e0219567ecd42a01a30b55aed438d6c55ab1d4a6d7ea62fe7a65ac461874f1f1c5de59 ./Spot-On

0cf523d52de5f50ec01e7f442c0f374ac15b6432757f17ac93df26f88784d5819f27101a10a6e765b4d7e8ca420b7c110e822dc835ad0083b339b64f1431ec81 ./Spot-On-Kernel

date && sha512sum ./Spot-On ./Spot-On-Kernel

Fri Jan 6 10:28:47 2017

0fbcf8e8a82dacd9825c42cf88d2d5a70d87965da20a3bfc5f87b8f634e0219567ecd42a01a30b55aed438d6c55ab1d4a6d7ea62fe7a65ac461874f1f1c5de59 ./Spot-On

0cf523d52de5f50ec01e7f442c0f374ac15b6432757f17ac93df26f88784d5819f27101a10a6e765b4d7e8ca420b7c110e822dc835ad0083b339b64f1431ec81 ./Spot-On-Kernel

Non-digest-reproducible builds will generate distinct results.

date && sha512sum ./Spot-On ./Spot-On-Kernel

Fri Jan 6 10:30:43 2017

4ccc44773bd43086941dd26f02b82e2eca7972f768844a544d23c19fedf6ed407911a73af4493c546c9bf94ec60afd69c5e0a646e8f9378c75811f4b4d36ef9e ./Spot-On

90720de4702bd5df97ce1af6ec5fd8ec206b279fecff37200f5bad5849e4e1183131ef7492a3a94f45db648ea415184e1ab89da76e7ca9d493738de37e6a8bb5 ./Spot-On-Kernel

date && sha512sum ./Spot-On ./Spot-On-Kernel

Fri Jan 6 10:32:10 2017

ea6275a644d1cd589510907929d8133b9e4557abca8c19aa4523887d0efe2b084f24b80216e5b3c889d23d0071afbac1933b772bae5f7cdbb58bd383be4a4a8e ./Spot-On

51d59a7dccd39a34183eeb200e2658d3d77a24e4ca5facbb9ee95fb8f32dbb7434f2d354342c5d7fefd0104feeda45fad6a9fe33a81ee6cd5cf76619a44932e5 ./Spot-On-Kernel

Configuration Settings

Spot-On implements a defensive approach with respect to configuration settings. Shortly after the kernel and the user interface are started, important settings are reviewed and if necessary corrected. The potentially-adjusted values are stocked in global containers. Some methods also inspect critical values, adjusting them if necessary.

Constant-Time Comparison Function

Spot-On attempts to utilize constant-time byte comparison functions so as to avoid timing analysis. Comparisons that occur within database queries are not guaranteed to be constant-time.

DTLS

Spot-On supports DTLS over unicast. Qt version 5.12, or newer, is required.

Echo

Spot-On introduced the Echo. The Echo is a malleable concept. That is, an implementation does not require rigid details. Each model may adhere to their own peculiar obligations. The Echo functions on the elementary persuasion that information is dispersed over multiple or singular passages and channel endpoints evaluate the suitability of the received data. Because data may become intolerable, Spot-On implements its own congestion control algorithm. Received messages that meet some basic criteria are labeled and duplicates are discarded. Advanced models may define more sophisticated congestion-avoidance algorithms based upon their interpretations of the Echo.

Spot-On provides two modes of operation for the general Echo, Full Echo and Half Echo. The Full Echo permits absolute data flow. The Half Echo defines an agreement between two endpoints. Within this agreement, information from other endpoints is prohibited from traveling along the private channel.

Echo Public Key Sharing

The Echo Public Key Sharing construct was introduced by Mr. Schmidt. It is an elegant compliment to the Echo. The concept may be summarized as follows:

1. A community is created. The community is defined by a pair of authentication and encryption keys. The keys are derived via the PBKDF2 function.
2. Public key pairs may be optionally exchanged via the community. Participants who subscribe to a well-defined community will automatically accept public key pairs from participants who have published their public keys to the respective community.

Electronic Mail

Spot-On provides two e-mail models for distributed e-mail. Endpoints may optionally define themselves as institutions or post offices, or both.

A brief description of e-mail institutions follows. E-mail institutions are artificially characterized by addresses and names. The information is not considered secret and several endpoints may identify themselves identically. It is the responsibility of an institution to define subscribers. The data that an institution houses is stored in encrypted containers. Unlike physical institutions, Spot-On institutions are only allowed to read the signature portions of e-mail letters. The signatures allow verification of deposits and withdrawals.

One important difference between e-mail institutions and e-mail post offices is that post offices require the distribution of public keys.

Electronic Mail Forward Secrecy

This section briefly describes a two-step communication process for establishing forward secrecy in Spot-On e-mail. We assume a hybrid scheme with respect to public-key encryption.

Assumptions:

1. Permanent public key pairs have been exchanged correctly.
2. The respective kernels remain active during the exchange window.

Protocol:

1. Participant A generates an ephemeral public encryption key pair. The key pair's attributes are configurable. If the kernel is deactivated after the key pair is generated, the key pair is discarded and the protocol is terminated.
2. Participant A transmits the ephemeral public key to B. The key is encrypted with B's permanent encryption public key and optionally signed with A's permanent private signature key.
3. B receives the public key and optionally verifies the signature. If B requires a valid signature but one is not provided, the protocol is terminated.
4. B generates private authentication and encryption keys.
5. Using the ephemeral public encryption key, B transfers the keys to A. The complete bundle is encrypted with A's permanent encryption public key and optionally signed with B's permanent private signature key.
6. Participant A receives the bundle and optionally verifies the signature. If A requires a valid signature but one is not provided, the protocol is terminated.

The session keys generated in the fourth step may remain in use until one of the parties decides to establish new session keys.

Signatures are required over the Poptastic transport.

Encrypted and Authenticated Containers

Some of the data that Spot-On retains locally is stored in encrypted and authenticated containers. CBC and CTS encryption modes are used with a variety of block ciphers. Authentication and encryption occur as follows:

1. If the size of the original data is less than the specified cipher's block size, the original data is re-sized such that its new size is identical to the cipher's block size. A zero-byte pad is applied.
2. Append the size of the original data to the potentially-padded container.
3. Encrypt the augmented data via the selected cipher and specified mode.
4. Compute a keyed-hash of the encrypted container.
5. Concatenate the hash output with the encrypted data, $H_{\text{Hash Key}}(E_{\text{Encryption Key}}(\text{Data} \mid\mid \text{Random} \mid\mid \text{Size(Data)})) \mid\mid E_{\text{Encryption Key}}(\text{Data} \mid\mid \text{Random} \mid\mid \text{Size(Data)})$.

Spot-On also includes a mechanism for re-encoding data if new authentication and encryption keys are desired.

File Encryption

Spot-On includes a simple file encryption application. A PIN and secret are used for generating authentication and encryption keys. After the initial encryption key is derived and applied, a subsequent encryption key is derived via a single-round PBKDF2 using the current key as the secret and the encrypted data as the salt. Several encryption and hash algorithms are supported as well as various encryption modes.

The format of the converted file is as follows:

Byte 0: 0 – unsigned file, 1 – signed file	Bytes 1 – 64: keyed hash or 64 bytes of zeros	Bytes 65 – EOF: encrypted data

Fragmented StarBeams

Fragmented StarBeams allow for the fragmentation of a mosaic (file) into a number of N unique pulses, where N is the number of active network connections. The standard StarBeam transfers copies of a particular pulse over each network connection.

Key Derivation

Spot-On uses separate authentication and encryption keys for local data. The key-derivation process is as follows:

1. Generate a cryptographic salt. The size of the salt is configurable.
2. Derive a temporary key via the PBKDF2 function. The hash algorithm, iteration count, passphrase (question/answer), and salt are input parameters to the function. All of the aforementioned parameters are configurable.
3. Using the temporary key from the previous step, derive a new key via the PBKDF2 function. The previous parameters are also used, however, the temporary key replaces the passphrase (question/answer).
4. Separate the derived key into two distinct keys. The encryption key is N bytes long, where N is the recommended key size of the selected cipher. The remaining bytes compose the authentication key. The generated authentication key contains at least 512 bytes.

Hybrid System

Spot-On implements a hybrid system for authenticity and confidentiality. Per-message authentication and encryption keys are generated. The two keys are used for authenticating and encapsulating data. The two keys are encapsulated via the public-key portion of the system. The application also provides a mechanism for distributing session-like keys for data encapsulation. The private keys are encapsulated via the public-key system. An additional mechanism allows for the distribution of session-like keys via previously-established private keys. Digital signatures are optionally applied to the data. As an example, please consider the following message: $E_{\text{Public Key}}$(Encryption Key || Hash Key) || $E_{\text{Encryption Key}}$(Data) || $H_{\text{Hash Key}}$ ($E_{\text{Public Key}}$(Encryption Key || Hash Key) || $E_{\text{Encryption Key}}$(Data)). The private-key authentication and encryption mechanism is identical to the procedure discussed in the Encrypted and Authenticated Containers section.

Local Private Application Interfaces

Spot-On supports the concept of local private application interfaces. The interfaces allow networked applications to stream authenticated and encrypted data through a Spot-On network. Application-native cryptographic capabilities are not required. A local listener, such as 127.0.0.1:4710, should be defined per application. Once defined, credentials may be prepared for the listener. Let's review:

1. Decide on the interface of the application. That is, Bluetooth, TCP, etc. Does it require SSL/TLS?
2. Create a local private listener, say 127.0.0.1:4710.
3. Enable the pass-through setting on the listener.
4. Prepare the pseudo-private credentials via a context menu.
5. Distribute the credentials to your partners. Remember, these are not necessarily private credentials. However, let's consider them as such.
6. Initiate the Spot-On kernel process.
7. Connect your application to the previously-defined listener.

Do remember that if Spot-On is on a public network, data will arrive through the public interface(s). However, your networked application will only receive applicable data; that is, data that was encapsulated by your credentials.

Please note that listeners may be defined such that remote access is possible. For example, one may define 192.168.178.100:4710. A similar approach may be used to define private-application neighbors.

Some other interesting conclusions follow.

Adaptive echoes are not supported on private-application interfaces.

Applications may not be able to support Spot-On accounts. Therefore, Spot-On accounts are not observed for private-application listeners.

Echo modes and lane widths are regarded.

If congestion control becomes an issue, you may wish to prolong the congestion timer.

Spot-On guarantees ordered data delivery.

Strange behavior may occur if multiple applications share a common channel.

McEliece

Spot-On integrates an independent and self-contained classical McEliece implementation. The implementation is based on the software and writings of Antoon Bosselaers, René Govaerts, Robert McEliece, Bart Preneel, Marek Repka, Christopher Roering, Joos Vandewalle.

Some general information. Spot-On supports m value 11 and t value 51. For m = 11 and t = 51, k = 1487 and n = 2048. As a result, the message expansion factor is approximately 1.4. Parameters m = 12 and t = 68 are also provided.

A private keys consists of matrices P^{-1} and S^{-1}, the code support L, a binary irreducible Goppa polynomial g, and a vector. The matrices contain 2048 x 2048 and 1487 x 1487 entries, respectively. The polynomial contains 51 entries. The vector contains 2048 entries. A total of 6,407,572 entries are required. As many as 12,873,361 bytes may be consumed by a private key. Approximately 74 MiB are required for housing six McEliece private keys.

A public key consists of matrix $\hat{G}$ and t. A total of 1487 x 2048, or 3,045,376, entries are required. As many as 6,093,750 bytes are consumed.

Included is an interpretation, model a, of the Fujisaki-Okamoto conversion. Please see https://www.emsec.rub.de/media/attachments/files/2013/03/mastersthesis-hudde-code-based-cryptography-library.pdf for more details. The key streams referenced in the aforementioned paper are generated via single-round PBKDF2 and SHA-256. The generated 32-byte salts are transferred as clear text. Computation errors abort the processes.

Decryption

1. Decrypt, via McEliece, c_1 to obtain the original message m.
2. Compute the original error vector e via $e = c_1 – m * \hat{G}$.
3. Compute the SHA-256 digest of e.
4. Apply the previously-computed digest to a single round of the PBKDF2 function. The generated key stream, k_2, will contain 1488 bits of which the first 1487 will be consumed in the following computation. The 32-byte salt, s_2, is provided to PBKDF2.
5. Compute $mcar = c_2 \text{ xor } k_2$.
6. Compute the SHA-256 digest of e || mcar.
7. Apply the previously-computed digest to a single round of the PBKDF2 function. The generated key stream, k_1, will contain 1488 bits of which the first 1487 will be consumed in the following computation. The 32-byte salt, s_1, is provided to PBKDF2.
8. Verify that $c_1 = k_1 * \hat{G} + e$.

Encryption

1. Generate a random vector e of length n. The vector e will contain t randomly-dispersed ones.
2. Compute the SHA-256 digest of e || m, where m is the original message.
3. Apply the previously-computed digest to a single round of the PBKDF2 function. The generated key stream, k_1, will contain 1488 bits of which the first 1487 will be consumed in the following computation. A 32-byte weakly-derived salt, s_1, is provided to PBKDF2.

4. Compute $c_1 = k_1 * \hat{G} + e$.
5. Compute the SHA-256 digest of e.
6. Apply the previously-computed digest to a single round of the PBKDF2 function. The generated key stream, k_2, will contain 1488 bits of which the first 1487 will be consumed in the following computation. A 32-byte weakly-derived salt, s_2, is provided to PBKDF2.
7. Compute $c_2 = k_2$ xor m.
8. Transfer c_1, c_2, s_1, and s_2.

Included is an interpretation, model b, of the Fujisaki-Okamoto conversion. Please see https://www.emsec.rub.de/media/attachments/files/2013/03/mastersthesis-hudde-code-based-cryptography-library.pdf for more details. The key streams referenced in the aforementioned paper are generated via SHAKE-256. libgcrypt 1.7.0 or newer is required. Computation errors abort the processes.

Decryption

1. Decrypt, via McEliece, c_1 to obtain the original message m.
2. Compute the original error vector e via $e = c_1 – m * \hat{G}$.
3. Compute the SHAKE-256 digest of e. The generated key stream, k_2, will contain 1488 bits of which the first 1487 will be consumed in the following computation.
4. Compute mcar = c_2 xor k_2.
5. Compute the SHAKE-256 digest of e || mcar. The generated key stream, k_1, will contain 1488 bits of which the first 1487 will be consumed in the following computation.
6. Verify that $c_1 = k_1 * \hat{G} + e$.

Encryption

1. Generate a random vector e of length n. The vector e will contain t randomly-dispersed ones.
2. Compute the SHAKE-256 digest of e || m, where m is the original message. The generated key stream, k_1, will contain 1488 bits of which the first 1487 will be consumed in the following computation.
3. Compute $c_1 = k_1 * \hat{G} + e$.
4. Compute the SHAKE-256 digest of e. The generated key stream, k_2, will contain 1488 bits of which the first 1487 will be consumed in the following computation.
5. Compute $c_2 = k_2$ xor m.
6. Transfer c_1 and c_2.

Multicasting

UDP multicasting is available via neighbors. UDP multicast listeners are not necessary. Sample UDP multicast address: 239.255.43.21.

Multiple Devices

Spot-On allows identical representations of individual nodes across multiple devices. For example, one may configure a node on one device and copy the configuration to another device.

Non-Volatile Congestion Control Memory

Limited-memory devices may benefit from non-volatile congestion-control storage. Spot-On provides a mode where message digests are temporarily stored in a local SQLite database. Please see https://www.sqlite.org/wal.html and https://www.sqlite.org/pragma.html#pragma_synchronous.

NTL

The NTL library is required if the McEliece cryptographic system is desired. A modified form of the library's source is included in the source repository. The modifications include the removal of GMP support and the resolution of some compiler warnings. On OS X, Homebrew should be used to install the library and its dependencies. Generally, brew install ntl. If Spot-On is prepared via spot-on.qt5.pro, a shared form of the library will be prepared. The initial preparation process may require a significant time to complete. NTL source is available at http://www.shoup.net/ntl. Newer compilers may support NTL's exceptions implementation. Please see the DoConfig file. Please read the Documentation/MCELIECE document for additional information.

Pass-through Devices

Spot-On supports pass-through connections. Pass-through connections allow data travel without alterations. Please note that accounts are supported for such connections. All other data interactions, including capabilities announcements, cannot be supported.

Poptastic

Version 0.17 of Spot-On introduced Poptastic. The new mechanism allows participants to communicate via the IMAP, POP3, and SMTP protocols. Poptastic provides a medium for near real-time conversations as well as traditional e-mail. Please note that Poptastic does not support the two-way calling mechanism.

Problems

This section details some of the encountered problems that Spot-On exhibits.

- Account credentials are stored using reversible encryption.
- An abrupt disconnect from the URLs database is not acknowledged by a Spot-On user interface process.
- Automatic computations of StarBeam file digests may result in incorrect values.
- Surreptitious forwarding and e-mail messages.
- The Spot-On user interface may terminate abnormally during a query of the PostgreSQL URLs database. Although the function PQinitOpenSSL(0, 0) is referenced, the process may nonetheless terminate. OpenSSL 1.0.x. The kernel process may suffer a similar problem.
- The URL data stored in shared.db does not include digests. Uniqueness is not properly defined in the urls database table.

Public Key Infrastructure

Spot-On utilizes the libgcrypt and libntru libraries for permanent private and public key pairs. Presently, the application generates ten key pairs during the initialization process. Key generation is optional. Consequently, Spot-On does not require a public key infrastructure.

ElGamal, McEliece, NTRU, and RSA encryption algorithms are supported. DSA, ECDSA, EdDSA, ElGamal, and RSA signature algorithms are supported. The OAEP and PSS schemes are used with RSA encryption and RSA signing, respectively.

Communications between nodes having diverse key types are well-defined if the nodes share common libgcrypt and libntru libraries.

Non-McEliece and Non-NTRU private keys are evaluated for correctness via the gcry_pk_testkey() function. Public keys must also meet some basic criteria such as including the public-key identifier.

Public Key Sharing

In addition to the Echo Public Key Sharing, Spot-On offers a multitude of key-sharing methods. Keys may be shared via network connections. Keys may be exported to a system's clipboard buffer and imported via the user interface, that is, copy-and-paste. Keys may also be exported and imported via text files.

Public key pairs are validated whenever they are imported via the kernel and the user interface. As some cryptographic algorithms lack signature properties, keys inserted via network connections are temporarily accepted. Key-acceptance is optional. The Echo Public Key Sharing algorithm offers its own permissions.

Spot-On also includes a mechanism for exporting a specific key-pair using a specific participant's credentials, a so-called Repleo. A repleo is a public-key bundle which contains authentically-encrypted data. Only a designated participant will be able to process a repleo.

Secure Memory

Spot-On supports secure memory through libgcrypt. Both the kernel and the user interface processes may be configured to use secure memory. If secure memory is not necessary, simply set the appropriate setting to zero. Please note that operating systems may impose restrictions on the amount of memory that may be reserved. Please also note that Spot-On cannot guarantee the permanence of locked memory. Also, logical regions exist where private data are placed into QByteArray objects.

Sessions

Excluding Forward Secrecy, SMP, accounts, and congestion control, Spot-On does not maintain a deliberate state system with respect to communications. Some portions of the system provide the user with state information, however, the information is not required for the health and stability of Spot-On communications. Shared cryptographic data is maintained in authentically-encrypted containers.

Socialist Millionaire Protocol

As of version 0.19, Spot-On includes an asynchronous implementation of the Socialist Millionaire Protocol as defined by https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html. Spot-On uses the SHA-512 of the secrets as the x and y components. SHA-512 is also used during proof assembly and validation.

Assuming that Alice begins the exchange:

- Alice:
  1. Picks random exponents a_2 and a_3
  2. Sends Bob $g_{2a} = g_1^{a_2}$ and $g_{3a} = g_1^{a_3}$
- Bob:
  1. Picks random exponents b_2 and b_3
  2. Computes $g_{2b} = g_1^{b_2}$ and $g_{3b} = g_1^{b_3}$
  3. Computes $g_2 = g_{2a}^{b_2}$ and $g_3 = g_{3a}^{b_3}$
  4. Picks random exponent r
  5. Computes $P_b = g_3^r$ and $Q_b = g_1^r g_2^y$
  6. Sends Alice g_{2b}, g_{3b}, P_b and Q_b
- Alice:
  1. Computes $g_2 = g_{2b}^{a_2}$ and $g_3 = g_{3b}^{a_3}$
  2. Picks random exponent s
  3. Computes $P_a = g_3^s$ and $Q_a = g_1^s g_2^x$
  4. Computes $R_a = (Q_a / Q_b)^{a_3}$
  5. Sends Bob P_a, Q_a and R_a
- Bob:
  1. Computes $R_b = (Q_a / Q_b)^{b_3}$
  2. Computes $R_{ab} = R_a^{b_3}$
  3. Checks whether $R_{ab} == (P_a / P_b)$
  4. Sends Alice R_b
- Alice:
  1. Computes $R_{ab} = R_b^{a_3}$
  2. Checks whether $R_{ab} == (P_a / P_b)$
- If everything is done correctly, then R_{ab} should hold the value of (P_a / P_b) times $(g_2^{a_3 b_3})^{(x - y)}$, which means that the test at the end of the protocol will only succeed if x == y. Further, since $g_2^{a_3 b_3}$ is a random number not known to any party, if x is not equal to y, no other information is revealed.

Sources of Entropy

Spot-On nodes may be configured as sources of entropy. Private peers must export one or more connection points and enable the randomness-source mechanisms. Remote peers should then connect to the private peers via Half-Echo agreements. Be careful of malicious entropy!

Surreptitious Forwarding

The Spot-On e-mail methods of Spot-On are vulnerable to surreptitious forwarding. The errors will be addressed as resources permit. All other message types which may regard digital signatures include both the recipient’s and sender’s identities in the data that’s provided to digital-signature algorithms.

By including a recipient's identity alongside a sender's identity in a digital signature, surreptitious forwarding is discouraged. A digital signature is created from the concatenated data of message-specific data, symmetric-key data, the recipient’s identity, and the sender’s identity.

Two-Way Calling

Spot-On implements a plain two-pass key-distribution system. The protocol is defined as follows:

1. A peer generates 128-bit AES and 256-bit SHA-512 keys via the system's cryptographic random number generator.
2. Using the destination's public key, the peer encapsulates the two keys via the hybrid cryptographic system.
3. The destination peer receives the data, records it, and generates separate keys as in step 1.
4. The destination peer transmits the encapsulated keys to the originating peer as in step 2.

Once the protocol is executed, the two peers shall possess identical authentication and encryption keys. Please note that duplicate half-keys are allowed.

Verifying Ownership of Public Keys

A shared public key bundle includes the encryption public key, a signature of the encryption public key, the signature public key, and a signature of the signature public key. Ownership of McEliece and NTRU encryption keys is not verified. Signatures are generated as follows:

1. The encryption or signature public key is gathered and stored in *data_t*.
2. The SHA-512 digest of the public key is computed and stored in *hash*.
3. For DSA, Elliptic Curve DSA, and ElGamal public keys:
    1. gcry_sexp_build(&data_t, 0,

       "(data (flags raw)(value %m))",

       hash_t) is computed.
4. For Edward Curve DSA public keys:
    1. gcry_sexp_build(&data_t, 0,

       "(data (flags eddsa)(hash-algo sha512)"

       "(value %b))",

       hash.length(),

       hash.constData()) is computed.
5. For RSA public keys:
    1. gcry_sexp_build(&data_t, 0,

       "(data (flags pss)(hash sha512 %b)"

       "(random-override %b))",

       hash.length(),

       hash.constData(),

       random.length(),

       random.constData()) is computed, where the variable *random* contains 20 bytes of random data.
6. Finally, gcry_pk_sign(&signature_t, data_t, key_t) is computed. The contents of *signature_t* are extracted via gcry_sexp_sprint() and stored in a byte array. The variable *key_t* contains either the public encryption key or the public signature key.

Signatures are verified as follows:

1. The SHA-512 digest of the provided public key is computed and stored in *hash*.
2. For DSA, Elliptic Curve DSA, and ElGamal public keys:
    1. gcry_sexp_build(&data_t, 0,

       "(data (flags raw)(value %m))",

hash_t) is computed. The contents of *hash_t* are populated via gcry_mpi_scan().

3. For Edward Curve DSA public keys:
    1. gcry_sexp_build(&data_t, 0,

       "(data (flags eddsa)(hash-algo sha512)"

       "(value %b))",

       hash.length(),

       hash.constData()) is computed.
4. For RSA public keys:
    1. gcry_sexp_build(&data_t, 0,

       "(data (flags pss)(hash sha512 %b)"

       "(random-override %b))",

       hash.length(),

       hash.constData(),

       random.length(),

       random.constData()) is computed, where the variable *random* contains 20 bytes of random data.
5. Finally, gcry_pk_verify(signature_t, data_t, key_t) is computed. If gcry_pk_verify() returns zero, the signature is considered valid. The contents of *signature_t* are populated via gcry_sexp_new() using the provided signature array.

Wide Lanes

One of the many obligations of a Spot-On-Kernel process is to receive, process, and forward data to one or more nodes. The mechanism that performs this task is similar to both a network hub and a network switch. Wide Lanes allow node operators to assign listener lane widths. Let's consider a basic example, a listener having a lane width of 20,000 bytes. The kernel, if necessary, will forward packets via the listener's clients if the sizes of the forwarded packets do not exceed 20,000 bytes. Optionally, clients may negotiate different lane widths with their peers. All network communications beyond the interface and the kernel must and will adhere to the configured limits.

References

http://world.std.com/~dtd/sign_encrypt/sign_encrypt7.html

https://blog.cryptographyengineering.com/2012/02/02/multiple-encryption/

https://en.wikipedia.org/wiki/Authenticated_encryption

https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

https://en.wikipedia.org/wiki/Ciphertext_stealing

https://en.wikipedia.org/wiki/Hybrid_cryptosystem

https://en.wikipedia.org/wiki/NTRU

https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding

https://en.wikipedia.org/wiki/PKCS_1

https://en.wikipedia.org/wiki/Padding_%28cryptography%29

https://en.wikipedia.org/wiki/Stream_Control_Transmission_Protocol

https://en.wikipedia.org/wiki/Time-based_One-time_Password_Algorithm

https://en.wikipedia.org/wiki/Timing_attackhttps://en.wikipedia.org/wiki/Timing_attack

https://eprints.qut.edu.au/35665/1/c35665.pdf

https://gnupg.org/documentation/manuals/gcrypt

https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html

https://www.cs.jhu.edu/~astubble/dss/ae.pdf