Inky Security White Paper

This document describes security properties of the Inky system as of June 2016, as deployed for individuals and small- to medium-sized teams, where the overall trust anchor is a user-selected password.

Preliminaries

This section defines the nomenclature used in this document and the system components covered. We also provide some insight into our general security principles.

Nomenclature

This section defines terminology used in this document.

- *User* means a person using Inky as an individual or as part of a small group, such as a family or small- to medium-sized business.

- A *Hardware Security Module* (HSM) is a device that can securely store private keys and perform decryption and signing operations with these keys. Importantly, an HSM never reveals its stored keys to the host computer or application.

- *Inky Password* refers to the user's chosen Inky login password. The password is a *trust anchor* -- it must therefore be strong enough to resist brute force guessing, and must be kept secret. The Inky Password is only used to provision new devices; once a device is provisioned, access to the Inky Client is protected by the standard host operating system mechanisms (Windows logon, thumb print, etc.)

- *Inky Client* refers to the application software that enables sending and receiving messages using the Inky system. Inky Client implementations are available for iOS, Android, OS X, Windows, and Chrome OS.

- An *Inky Profile* is a collection of information that Inky maintains about an Inky user. Every Inky user has an associated Inky Profile, uniquely identified by an Inky Identity. The Inky Profile stores settings, passwords, key pairs, certificates, and all other information required for the user to send and receive messages using mail accounts. Inky Profiles are stored by the *Inky Profile Server.*

- An *Inky Identity* uniquely identifies a particular Inky Profile. For users, the Inky Identity is an ASCII userid selected by the user. For large enterprise users, the Inky Identity is namespaced (e.g., `customer/user`).

- An *Inky Identity Key* is a symmetric key used to encrypt sensitive data in the Inky Profile. It is a 64 byte random value generated client-side when the Inky Profile is created.

- An *Inky Identity KEK* is a 64-byte key derived from the user's Inky Identity, Inky Password, and Profile Salt, via a key derivation algorithm. (*KEK* is an abbreviation for *Key Encryption Key.*)

- A *Profile Salt* is a random value generated client-side when the Inky Profile is created.

- A *Verifier* is a server-side cryptographic object used for zero-knowledge-proof-based authentication, and derived from some client secret.

  An important property of the verifier is that, under standard assumptions of cryptography, it is impossible to derive the client secret from the verifier object any more efficiently than via a brute-force dictionary attack.

- An *Inky Verifier* is a Verifier derived from the Inky Identity and the Inky Password; it facilitates authentication of the Inky Client to the Inky Profile Server.

- The *Inky Identity Server Private Key Signing Key* is an elliptic curve key pair. The *Inky Identity Server Private Key Encryption Key* is an RSA key pair. Together, these are used to store Inky Identity Keys securely in the cloud. The private key halves of these key pairs only exist on HSMs. See *Secure Inky Identity Key Storage* below for details.

- *S/MIME* refers to a set of X.509/PKI-based standards, established via internet RFCs, governing the exchange of digitally signed and/or encrypted emails. Use of the S/MIME standards is mandated for US Government classified email. The Inky Client creates and interprets signed and encrypted emails in accordance with the S/MIME standards.

The `||` symbol denotes string concatenation throughout this document.

Inky System Components

This section describes the main client and server components of the Inky system.

- The *Inky Client* component runs as an installed application on supported devices. It acts as a TCP/IP and UDP network client for numerous services: DNS, IMAP, SMTP, POP, and Exchange services, web services (geocoding, package tracking, etc.), and the Inky Profile Server. It never acts as a network server itself. It runs as a single system process.

- The *Inky Profile Server* component runs as a load-balanced server in Inky's virtualized AWS cloud. It acts as a network server, providing profile information -- user settings as well as encrypted passwords and keys -- to authenticated Inky Clients. The Inky Profile Server uses the Inky Identity Server and Inky Key Server to provide (indirect) access to these services to the Inky Client.

- The *Inky Identity Server* component runs as a load-balanced server in Inky data centers, with physically attached HSM devices containing private encryption and signing keys for Inky Identity Keys. This server maintains a simple database mapping each Inky Identity to an encrypted copy of the corresponding Inky Identity Key.

  The Inky Identity Server also acts as a Certificate Authority, accepting Certificate Signing Requests from the Inky Profile Server and issuing new S/MIME certificates on behalf of users. The Inky Identity Server only accepts connections from the Inky Profile Server.

- The *Inky Key Server* component runs as a load-balanced server in Inky's virtualized AWS cloud. It stores public key certificates for all Inky users and supports querying for certificates by email address. The Inky Key Server only accepts connections from the Inky Profile Server.

- The *Inky Email Verification Server* component runs as a load-balanced server in Inky's virtualized AWS cloud. It receives verifcation emails sent from the Inky Client and replies with S/MIME signed reply emails.

Note that the Inky Client interoperates with potentially many other servers -- for example, to retrieve the user's email. We limit the protocols used to communicate with these servers, but do not consider these servers to be part of the Inky system, or within its notional security perimeter.

Inky Security: Guiding Principles

Throughout the design and implementation of the Inky system, we have adhered to the following general security principles:

- We use only strong, authenticated symmetric ciphers: generally AES-256-GCM. For whole-database encryption on slower devices we use AES-128-GCM.

- We use only strong asymmetric encryption: 4096-bit RSA where supported; otherwise 2048-bit RSA.

- We use only elliptic curve cryptography for signing -- specifically the P-256 curve, because it is strong enough, has an assembly language accelerator available in OpenSSL, and is well supported by other cryptographic libraries.

- When we control both client and server, we use only TLS 1.2 with the strongest ciphersuites available -- in particular, those that offer perfect forward secrecy: `ECDHE-RSA-AES256-GCM-SHA384` and `ECDHE-ECDSA-AES256-GCM-SHA384`.

- When we control only the client -- for example, when connecting to a mail server -- we negotiate the best supported TLS version and ciphersuite, requiring at least SSLv3. (The minimum allowed protocol is configurable.)

- For authenticated ciphers like AES-GCM, we ensure generated IVs are both unique and random, by incorporating a time stamp.
- We use only SHA-256 and SHA-512 digest algorithms.
- We perform complete X.509 certificate verification per RFC 5280, and implement revocation checking via CRLs, OCSP, and OCSP stapling.
- We use the zxcvbn open source package to estimate user password strength, and require passwords that are nontrivial to brute force.
- We employ code-signing to prevent tampering with client code. `eval` and `exec` are disallowed in Python code. Except in our JavaScript UIs we only execute code that has been code-signed. All cryptographic operations are executed by signed code; in particular, this means we never perform any cryptographically relevant operations in JavaScript.
- We sanitize all HTML content before display.
- Server-side SQL queries never incorporate user-supplied text other than the Inky Identity itself, which may not contain spaces.
- We store *all* data encrypted at rest on endpoint devices. Full text search indices are stored encrypted as well.
- Sensitive values such as private keys and passwords are stored "whitebox encrypted" in memory using AES-256-GCM. An encrypted will only ever exist in plaintext from transiently, for the single operation is is needed for, and will be cleansed (zero) after use. This ensures sensitive values do not remain in memory, and are not (easily) accessible via core dumping the process, etc.
- We use only standard cryptographic primitives: well-known standard digest algorithms, ciphers, and key derivation schemes. We use standard CMS containers to encapsulate signed and encrypted data, and implement the S/MIME standard for signed and encrypted email.
- We stay up to date with upstream changes to OpenSSL, the cryptographic library we use.

Security-Related System Component Interactions

This section describes interactions between components of the Inky system with security implications.

Inky Profile Creation

To create a new profile, a user first downloads the Inky application. On first startup, the application offers two options: "sign in" (provision) and "register". To create a new profile, the user clicks "register". The user can then choose an ASCII Inky Identity.

The Inky Client sends a message requesting the new userid to the Inky Profile Server. If an Inky Profile already exists for this Inky Identity, the server tells the client to ask the user to pick a different Inky Identity. Once the user selects a valid new Inky Identity, the Inky Profile Server responds accordingly.

The user can then choose an Inky Password. Password strength is estimated with the zxcvbn open source package and is displayed to the user. The user must select a suitably strong password to proceed.

Once the user has chosen an acceptable Inky Identity and Inky Password, the Inky Client creates two cryptographically random values:

- The Profile Salt
- The Inky Identity Key

The Inky Client then derives the Inky Identity KEK from the Inky Identity, Inky Password, and Profile Salt, using the derivation

```
inky-identity-kek = PKCS5_PBKDF2_HMAC(inky-identity || inky-password, profile-salt)
```

where the generated key length is 64 bytes, the digest algorithm used is SHA-512, and the number of iterations is 65536.

The Inky Client then uses the Inky Identity KEK to encrypt the Inky Identity Key it just generated. See *Inky Profile Value Encryption* for details; the difference from the process described there is the key used, since obviously the Inky Identity Key cannot be used to encrypt itself.

At this point the Inky Client also derives the Inky Verifier from the Inky Identity and the Inky Password; this process is described below in *Inky Profile Server Authentication*.

The Inky Client then sends the Inky Verifier, the encrypted Inky Identity Key, and the Profile Salt to the Inky Profile Server. The Inky Profile Server creates a new, default profile for the user that includes the Profile Salt. It then associates the Inky Verifier with the new profile. This establishes the basis for future authentications from the Inky Client to the Inky Profile Server (until the user changes his/her password).

Finally, the Inky Profile Server sends the encrypted Inky Identity Key for the new user to the Inky Identity Server. The Inky Identity server then further encrypts the Inky Identity Key and stores it, as described in *Secure Inky Identity Key Storage*. If the encryption and storage of the (now doubly-encrypted) Inky Identity Key succeeds, the Inky Identity Server reports success back to the Inky Profile Server, which in turn reports success back to the Inky Client.

The Inky Client then returns the user to the register/sign in screen, where the user can provision the device as usual.

Provisioning New Devices

Once an Inky Profile has been created for a user by the Inky Profile Server, the user can self-provision *any* new device for that profile by correctly entering the Inky Identity and Inky Password. Provisioning only needs to be done once on each device.

To provision, the user selects "sign in" from the application launch screen and enters the Inky Identity and Inky Password. The Inky Client then derives the Inky Verifier and uses it to authenticate to the Inky Identity Server as described in *Inky Profile Server Authentication*.

Once authenticated, the Inky Client sends a provisioning request to the Inky Profile Server. The Inky Profile Server will then ask the Inky Identity Server to decrypt the Inky Identity Key for this user, and, upon receiving it, will return it to the Inky Client. (Recall that at this point the Inky Identity Key is still encrypted with the Inky Identity KEK derived from the user's Inky Identity and Inky Password, so neither the Inky Identity Server nor the Inky Profile Server can fully decrypt it.)

The Inky Client then decrypts the Inky Identity Key using the Inky Identity KEK, and proceeds to retrieve profile information from the Inky Profile Server, using the Inky Identity Key to decrypt sensitive values when needed. Once the Inky Client has all the profile data, it can connect to mail servers and retrieve mail, etc., using credentials it decrypts with the Inky Identity Key.

Finally, the Inky Client stores the Inky Identity Key on the local file system, still encrypted with the Inky Identity KEK. This obviates the need to send a provisioning request to retrieve the Inky Identity Key at every startup.

Synchronization of Inky Profile Changes

The Inky Profile stores settings information as well as secrets like keys, passwords, and authentication tokens; it also includes so-called "annotations" to individual messages and email senders. Taken altogether, a single user's Inky Profile can include thousands of key/value pairs.

These key/value pairs are synchronized between Inky Client instances and the Inky Profile Server "in the background". The synchronization mechanism relies on a last modified date stored for each key/value pair. In addition, every key/value pair has an associated minimum schema level, ensuring that clients only receive updates to profile elements they can understand.

Profile key/value synchronization works identically for encrypted and plaintext values; the encrypted values are simply synchronized as ciphertext. Every field in the Inky Profile schema has a datum type (e.g., "string") and a set of properties (e.g., "encrypted"), and the Inky Client and Inky Profile Server both enforce type checking on all keys and values.

Public Key Storage and Retrieval

Key pairs for signing emails are always generated client side; the Inky Client asks the Inky Key Server to store, or "publish" the public key half of each pair it generates for email signing.

Public keys are not sensitive data -- they are public, after all -- so we simply store them as PEM data in a SQL database. Note, however, that only the Inky Profile Server has connectivity to the Inky Key Server, and all queries for public keys must be intermediated through the Inky Profile Server.

S/MIME as Implemented by the Inky Client

The Inky Client implements the S/MIME standard, as defined in RFC 5751, with elliptic curve cryptography support (defined in RFC 5753) and Triple Wrapping (defined in RFC 2634).

When encrypting outbound email, the Inky Client uses the AES-256-CBC symmetric cipher. When signing outbound email, the Inky Client uses ECDSA with the standard NIST P-256 curve.

Inbound email generated by clients other than the Inky Client is interpreted subject to policy controls. The default policy treats the following ciphers and digest algorithms as secure for incoming email:

```
AES-256-CBC
AES-192-CBC
AES-128-CBC
DES-EDE3-CBC
SHA-512
SHA-1
SHA-384
SHA-256
SHA-224
```

Every standard elliptic curve is considered valid and secure.

Note that standard S/MIME does not support the use of authenticated symmetric ciphers, and the RFC covering AES-GCM support (RFC 5084) mandates the use of the AuthEnvelopedData Cryptographic Message Syntax container, which is not widely supported.

Since we endeavor to use only authenticated symmetric ciphers -- see *Inky Security: Guiding Principles* above -- this is a serious limitation of the standard for us.

However, Triple Wrapping allows to work around this problem: for every outbound encrypted email, we create a signed/encrypted/signed CMS hierarchy. As explained in RFC 2634,

> The inside signature is used for content integrity, non-repudiation with proof of origin, and binding attributes (such as a security label) to the original content. These attributes go from the originator to the recipient, regardless of the number of intermediate entities such as mail list agents that process the message. The signed attributes can be used for access control to the inner body. Requests for signed receipts by the originator are carried in the inside signature as well.
>
> The encrypted body provides confidentiality, including confidentiality of the attributes that are carried in the inside signature.
>
> The outside signature provides authentication and integrity for information that is processed hop-by-hop, where each hop is an intermediate entity such as a mail list agent. The outer signature binds attributes (such as a security label) to the encrypted body. These attributes can be used for access control and routing decisions.

This means we can safely use AES-CBC as the symmetric cipher for the encrypted container, because it's authenticated by the signed outer container.

Every email sent with the Inky Client is signed using the certificate associated with the Inky user and the originating email address. For convenience, outbound emails that are *only* signed (not also encrypted) are sent "clear-signed", as described in RFC 5751, Section 3.4.3. This allows recipients lacking S/MIME processing capabilities to view the message.

Outbound encrypted emails are always triple-wrapped, as described above.

Verification of Email Accounts for S/MIME

We act as a Certificate Authority, issuing S/MIME signing and encipherment certificates to Inky users. This section describes the process we follow to verify that an email account is controlled by a user before we issue S/MIME certificates for that email account to that user.

To understand why email account verification is important, consider Mallory, an attacker. Mallory cleverly installs an email server that calls itself whitehouse.gov, and changes the DNS on his local network to redirect imap.whitehouse.gov and smtp.whitehouse.gov to his new server. Mallory can now add the email account president@whitehouse.gov to Inky, as long as he runs the Inky Client on a local machine with the modified DNS records.

Of course, we don't want Mallory to be able to create S/MIME certificates for president@whitehouse.gov, because Mallory doesn't control the *real* president@whitehouse.gov account.

The solution is to escape any DNS tampering Mallory might be doing by sending an email from our server infrastructure, where we control the DNS. The remainder of this section describes the verification process in detail. We'll assume for purposes of discussion that we are verifying the email address user@example.com.

The Inky Client begins the verification process by authenticating to the user's mail server for user@example.com and sending a verification email *from* user@exmample.com. This essentially mimics what happens when the user manually composes an email and sends it from user@example.com. The verification email contains the email address being verified, the related Inky Identity, and a random code.

The Inky Email Identification Server receives each email sent to verify@verify.inky.com. It replies to each sender with a new email that includes, as an attachment, the RFC 2822 text of the received email -- i.e., the email the Inky Client sent on the user's behalf. Critically, the Inky Email Identification Server signs the mail it sends using S/MIME, using its private key.

The Inky Client waits for the mail from the Inky Email Identification Server to be delivered. Once it's been delivered and retrieved by the Inky Client, the Inky Client parses the email and ensures the random code matches. (Note that the random code serves no security purpose; it just makes it easier for the Inky Client to find the right verification email.)

The Inky Client then sends the signed verification email to the Inky Profile server, which in turn sends it to the Inky Identity Server. (Note that these send operations are via the dedicated TLS connections to these servers, not via SMTP.)

The Inky Identity Server then verifies that the mail is properly signed by the Inky Email Verification Server, and that the signed contents include the expected Inky Identity and email address.

Assuming everything checks out, the Inky Identity server stores a signed "proof" of verification in its database. This is a CMS container signed with a private key stored only on an attached HSM, and containing the verification time, the verified email address, the Inky Identity, and the RFC 2822 text of the signed email from the Inky Email Verification Server.

Once the user's control of the email account has been verified, the Inky Client asks the Inky Profile Server (which in turn asks the Inky Identity Server) to issue the new S/MIME certificates. The Inky Identity Server then checks that the email accounts for which S/MIME certificates are being requested have associated (and correct) proof of verification. If no properly signed verification records exist for the userid/email address pair, the request fails. Otherwise, the certificates are issued and the PEM text returned to the client.

Details of Cryptographic Processes

Our goal for this section is to detail critical cryptographic components in the Inky system sufficiently for a practitioner to implement them.

Secure Inky Identity Key Storage

This section details how the Inky Identity Server securely stores users' private Inky Identity Keys. Recall that the Inky Identity Key is the trust anchor for *all* of a user's Inky Profile data, including passwords, keys, authentication tokens, etc. So it is imperative that we store it securely.

Each private key is stored as a standard Cryptographic Message Syntax (CMS) container, as described by RFC 5652. The container is triply nested, with an outer signed container, a middle encrypted container, and an innermost signed container. We use this sign/encrypt/sign method to ensure we're only ever attempting to decrypt authenticated ciphertext.

We use separate cert/key pairs for signing and encryption:

- Inky Identity Server Private Key Encryption Certificate (G1)
- Inky Identity Server Private Key Signing Certificate (G1)

These certificates are both signed by this chain:

- Inky Root CA Certificate (G1)
- ∟ Inky Identity Server Root Certificate (G1)
-     ∟ Inky Identity Server Private Key Encryption Certificate (G1)
-     ∟ Inky Identity Server Private Key Signing Certificate (G1)

The signing certificate holds an elliptic curve public key based on the NIST standard P-256 curve. The encryption certificate holds a 2048-bit RSA public key. The private keys corresponding to all the certificates above are only ever stored on hardware security modules.

Thus decryption of a user's Inky Identity Key requires a properly provisioned HSM device *and* the user's Inky Identity KEK -- the latter of which is derived from the user's password which only the user knows.

Inky Profile Value Encryption

Most values in a user's Inky Profile are stored unencrypted, because they are not sensitive. These are settings that affect the user interface, notification choices, and so on.

Sensitive settings, such as email passwords, key pairs, oAuth tokens, and so on are kept encrypted at all times (except transiently, at the exact moment of use of the plaintext by the Inky Client).

The Inky Client encrypts sensitive settings using AES-256-GCM authenticated symmetric encryption. The keying material is the user's Inky Identity Key concatenated with the account salt. The key is simply the first N bits of the SHA-512 hash of the keying material, where N is the required key length for the cipher. (Defining it this way facilitates cipher agility.)

The IV is randomly generated and incorporates the current time to ensure lifetime uniqueness. The specific formulation of the IV is:

```
timestamp = UTC-time-since-epoch-in-seconds & 0xFFFFFFFF
IV = chr((timestamp >> 24) & 0xFF) ||
     chr((timestamp >> 16) & 0xFF) ||
     chr((timestamp >> 8) & 0xFF) ||
     chr(timestamp & 0xFF) ||
     random-bytes
```

Intuitively, this sets the first 4 bytes of the IV to the current time in seconds since the epoch UTC, and randomly sets the remaining bytes.

The encrypted result is then stored, along with the IV and the name of the cipher. All these values are base64-encoded.

Each encrypted value is kept in this form in memory until the moment it is needed. At that point, the value is decrypted and the plaintext is used. For example, if the plaintext is a password, the password might be written on a TLS socket connection to a mail server during authentication. Immediately after use, the plaintext is zeroed and discarded, ensuring decrypted values do not remain in memory.

Representation of Private Keys Within the Inky Profile

The Inky Profile Server stores the private keys used to sign email from the user, and to decrypt email to the user. These keys are standards-based: P-256 elliptic curve keys for signing, and 4096-bit RSA keys for encipherment.

Each pair (public key certificate and private key) is stored -- along with any additional chain certificates required to verify the certificate -- in a PKCS#12 container encrypted with a 16-byte random password.

The PEM text and the password for the PKCS#12 container are both stored encrypted in the user's Inky Profile. (See *Inky Profile Value Encryption*).

Private keys are always kept wrapped in memory. The wrapping is "whitebox" encryption using AES-256-GCM with a random key and IV. While this provides no additional security from a theoretical standpoint, it makes recovering keys by core dumping the running Inky Client process more difficult.

Inky Profile Server Authentication

The Inky Client uses only long-lived TLS connections to communicate with the Inky Profile Server. Both TLS peers must provide the correct (pinned) certificate for the connection to succeed. The TLS ciphersuite is restricted to either of

`ECDHE-RSA-AES256-GCM-SHA384`

or

`ECDHE-ECDSA-AES256-GCM-SHA384`

and TLS version 1.2 or higher is required. The elliptic curve must be P-256.

As long as an authenticated connection remains established, the server assumes the client is properly authenticated; no session cookies or other authentication-related state values are transmitted. However, once a connection is torn down for whatever reason, the client must authenticate again over a newly-established TLS session.

Authentication from the Inky Client to the Inky Profile Server is done via a password-authenticated key agreement (PAKE) protocol designed by Tom Wu called SRP, described (somewhat vaguely) in RFC 2945.

We deviate from the RFC slightly -- by using SHA-512 as the hash function, and by including additional information in the client authenticator. Our implementation closely follows Boyd & Mathuria, *Protocols for Authentication and Key Establishment*, p.264, and we will use their notation here:

`inky-identity` : the user's Inky user name (a string)

`inky-password` : the user's Inky password (a string)

`salt` : 32 random bytes stored with the verifier on the server

A: the client

B: the server

p: large prime such that $p = 2q + 1$

Z_p^* : the multiplicative group of nonzero integers mod p

G: subgroup of Z_p^* of order q, $G = \{n | n = g^k \bmod p, 1 <= k <= q\}$

g: a primitive root mod p

r_A: random integer chosen by A: $1 <= r_A <= q$

r_B: random integer chosen by B: $1 <= r_B <= q$

u: random integer chosen by B of the form $u = g^r \bmod p$ $(u \neq 0)$

t_A: public random value derived from r_A

K_{AB}: shared key negotiated by A and B

For p and g, we use the 2048-bit `MODP` group described in RFC 3526. The random r_A, r_B, and u values are all chosen to be 256 bits long.

The derivation of the Inky Verifier V (which the Inky Profile Server stores) is:

h = `string-to-integer(SHA512(salt + SHA512(inky-identity || inky-password)))`

$V = g^h \bmod p$

`inky-verifier = base64(integer-to-string(V)) ':' base64(salt)`

The `string-to-integer` conversion is exactly as described in RFC 2945:

`i = S[n-1] + 256 * S[n-2] + 256^2 * S[n-3] + ... + 256^(n-1) * S[0]`

and the `integer-to-string` conversion is its exact inverse. The base64 function is the `base64url` encoding described in RFC 4648.

The authentication process involves the derivation of an integer shared key K_{AB} in a manner similar to Diffie-Hellman key exchange. Each side then computes two authenticators to mutually authenticate each other. The client authenticator is:

```
CA = HMAC-SHA-512(K_AB,
                  SHA512(integer-to-string(p)) ||
                  SHA512(integer-to-string(g)) ||
                  SHA512(inky-identity) ||
                  integer-as-string(S) ||
                  integer-as-string(t_A) ||
                  salt ||
                  integer-as-string(u))
```

Where `S` is the scrambled password verifier:

$S = V + g^{r_B} \bmod p \ (S \neq 0)$

and integer-as-string simply converts each integer to a literal string of decimal digits; for example,

`integer-as-string(197) = "197"`

The server's authenticator is:

`SA = HMAC-SHA-512(K_AB, integer-as-string(t_A) || CA)`

The specific protocol exchange is exactly as described in Boyd & Mathuria, p.264.

Inky Client Data-at-Rest Encryption

The Inky Client uses the popular public domain database SQLite to securely store data at rest. The version of SQLite we use is modified with patches derived from the Zetetic SQLCipher project; these patches implement page-level encryption of the database.

In our specific implementation, each page is encrypted using either AES-256-GCM or AES-128-GCM (the latter only on slower devices like Chromebooks). Keying material for the database is derived from the Inky Identity and Inky Password -- similarly to the Inky Identity KEK, though with a different random salt value.

Each page is separately encrypted with a unique, random IV. IVs are generated in a manner similar to the time stamp-based derivation described in *Inky Profile Value Encryption* above. In this case, however, a process-global 8-byte sequence number is set at startup time to the number to UTC seconds since the epoch, and this sequence number is simply incremented for each encrypted page. The remaining bytes of each IV are chosen randomly.

The database page number is included in the AES-GCM authenticated data, ensuring that pages may not be reordered.

The Inky Client uses SQLite's FTS4 full-text indexing extension to implement message search capabilities. Importantly, page-level encryption of the database ensures that both the message data and the indices are encrypted on disk at all times.

HTML Sanitization

From a security standpoint it's critical to "sanitize" HTML before displaying it. Sanitization removes JavaScript and other executable code, and prevents a variety of related exploits. Sanitizing HTML properly is difficult, because HTML markup is quite rich, and many exploits have been developed. For example, proper sanitization requires parsing not only the HTML tags, but the CSS markup as well. (And, obviously, the sanitizer must preserve the intended look of the original message as much as possible.)

The Inky sanitzer is written in C using libxml2. It is based on the concept of whitelisting: only attribute/element combinations known to be safe are allowed; all others are either stripped or converted to safe attribute/element replacments.

The Inky sanitizer passes the excellent cross-site scripting (XSS) test suite provided with the PHP HTMLPurifier project.

We also regularly test Inky against Mike Cardwell's Email Privacy Tester (https://emailprivacytester.com/).

Inky Client Hardening

We use code-signing to prevent tampering with the Inky Client code. The executable code for the application is code-signed using the standard mechanism for the host operating system. This ensures that if the executable code is modified, the operating system will refuse to run the application.

Much of the Inky Client code is written in Python, a dynamic, interpreted language based on bytecode. This bytecode is also code-signed. Each Python bytecode file is a signed container in Cryptographic Message Syntax format, as described in RFC 5652. The container is signed with a private key known only to the build manager responsible for production builds for a particular operating system.

We embed a modified Python interpreter into the Inky Client; this interpreter only executes bytecode that is signed with a valid key. This ensures that if the Python bytecode is modified, the Python interpreter, which is itself code-signed executable code, will refuse to run it.

In the handful of places where the Python standard library (or our own code) requires the execution of programmatically generated bytecode -- for example in the standard library's `namedtuple` implementation -- the generated (unsigned) code is executed only via signed executable code which has been carefully constructed to avoid providing a backdoor to execution of arbitrary Python source code. As well, the executable code must provide the flag `PyCF_ALLOW_UNSIGNED_EXECUTION` to the Python interpreter function (usually `PyRun_StringFlags`).

System Keys Pairs and their Functions

This section lists all the key pairs used in the Inky system.

HSM-based keys

- *Inky CA Root Key Pair (EC-256)*

  Root Certificate Authority key pair; trust anchor for our certificates, and used to sign certificates for all the HSM-based key pairs.

- *Inky Email Encryption Root Key Pair (EC-256)*
- *Inky Email Signing Root Key Pair (EC-256)*

  Used to sign email signing and encipherment certificates for Inky users.

- *Inky Identity Server Root Key Pair (EC-256)*

  Used to sign Inky Identity Server TLS certificates.

- *Inky Identity Server Private Key Signing Key Pair (EC-256)*
- *Inky Identity Server Private Key Encryption Key Pair (EC-256)*

  Used by the Inky Identity Server to sign and encrypt Inky Identity Keys.

- *Inky Email Verification Server Email Signing Key (EC-256)*

  Used by the Inky Email Verification Server to sign verification emails.

- *Inky Identity Server Verification Information Signing Key (EC-256)*

  Used by the Inky Identity Server to sign stored email account verification information.

- *Inky Profile Server Root Key Pair (EC-256)*

  Used to sign Inky Profile Server TLS certificates

- *Inky Key Server Root Key Pair (EC-256)*

  Used to sign Inky Key Server TLS certificates

- *Inky Code Signing Root Key Pair (EC-256)*

  Used to sign Inky code signing and bytecode signing certificates

- *Inky License Signing Root Key Pair (EC-256)*

  Used to sign Inky licenses.

Other keys

- *Inky Identity Server TLS Client Key Pair (EC-256)*
- *Inky Identity Server TLS Server Key Pair (EC-256)*
- *Inky Profile Server TLS Client Key Pair (EC-256)*
- *Inky Profile Server TLS Server Key Pair (EC-256)*
- *Inky Key Server TLS Client Key Pair (EC-256)*
- *Inky Key Server TLS Server Key Pair (EC-256)*

Used for peer validation in TLS connections.

Hardware Cryptographic Acceleration

On devices with Intel CPUs supporting AES-NI instructions, both the Inky Client and our server process will use these instructions to accelerate symmetric cipher operations. Likewise, on devices with ARM NEON Crypto instructions, the Inky Client will use these instructions. We presently only run server processes on Intel CPUs.

Pseudorandom Number Generation

For *all* cryptographically relevant random number generation, both the Inky Client and our server processes use the OpenSSL `RAND_bytes` function. On devices with Intel processors offering the RDRAND instruction, we have configured `RAND_bytes` to dispatch to a function using RDRAND. On all other devices, OpenSSL's own `RAND_bytes` implementation will be used; this ultimately calls `ssleay_rand_bytes`. This function, in turn, implements Hash_DRBG as described in NIST SP 800-90A DRBG, using SHA-1 as the hash function.

See

```
https://wiki.openssl.org/index.php/Random_Numbers
```

for more details.

Comments, Questions, or Concerns?

Please visit

```
http://inky.com/security
```

to provide feedback, or to ask questions -- or with information about potential weaknesses, vulnerabilities, or oversights in anything described in this document.