<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Personal Security Devices</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>The Master Seed</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>HD Key Generation</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Applications of HD Trees</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Application Isolation</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>Introduction</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>Overview</td>
<td>19</td>
</tr>
<tr>
<td>9</td>
<td>BOLOS Features</td>
<td>21</td>
</tr>
<tr>
<td>10</td>
<td>Hardware Architecture</td>
<td>27</td>
</tr>
<tr>
<td>11</td>
<td>Application Environment</td>
<td>31</td>
</tr>
<tr>
<td>12</td>
<td>Introduction</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>Getting Started</td>
<td>35</td>
</tr>
<tr>
<td>14</td>
<td>Interaction Between BOLOS and Apps</td>
<td>37</td>
</tr>
<tr>
<td>15</td>
<td>Application Structure and I/O</td>
<td>39</td>
</tr>
<tr>
<td>16</td>
<td>Display Management</td>
<td>41</td>
</tr>
<tr>
<td>17</td>
<td>Persistent Storage and PIC</td>
<td>43</td>
</tr>
<tr>
<td>18</td>
<td>Memory alignment</td>
<td>47</td>
</tr>
<tr>
<td>19</td>
<td>Common Pitfalls and Troubleshooting</td>
<td>53</td>
</tr>
<tr>
<td>20</td>
<td>Application Debug</td>
<td>57</td>
</tr>
</tbody>
</table>
21 Publishing an Application 61
22 External Documentation 71
23 Developing Secure Ledger Apps 73
Index 87
Note: This documentation is intended for developers only and not general users.

Ledger produces personal security devices such as the Ledger Nano S and the Ledger Blue, both of which are architected around a Secure Element and the BOLOS platform.

This documentation contains information about developing apps for these devices, from high-level concepts like hierarchical deterministic key generation to low-level details about the hardware architecture of these devices.
CHAPTER 1

Introduction

In this chapter, we’ll first provide some background information about personal security devices and hierarchical deterministic wallets. Then, we’ll introduce the application isolation technology that helps make Ledger devices secure and open-source friendly.
Personal security devices are designed to isolate cryptographic secrets (like PGP or Bitcoin private keys) from your potentially insecure computer which has known vulnerabilities. Storing cryptographic secrets on a physical medium that cannot be infected with a virus such as writing them on a piece of paper (or storing them on an encrypted drive) is secure until you need to use your funds. The instant you need to transfer your funds, you need to load your private keys onto your computer to sign the transaction and as such you expose them to potential malware (unless you’d like to perform the cryptographic operations necessary to sign the transaction by hand, which is far from convenient, to say the least). This is where personal security devices come in - they don’t just store your cryptographic secrets safely, they also allow you to perform operations with them (like signing transactions) securely and conveniently. Ledger devices also generate secrets securely with a large amount of entropy using an AIS-31 compliant true random number generator (TRNG). As such, it is more secure to use the device’s internally generated secrets rather than importing a secret from elsewhere which could have been compromised before being loaded onto the device.

Ledger leverages Secure Element technology to build personal security devices for cryptocurrencies and blockchains which provide an interface between humans and the blockchain world. They keep your private keys secure from hackers by storing them in a tamper-proof and eavesdropping-proof Secure Element. Additionally, the Ledger Nano S and Ledger Blue have a screen which serves as a trusted source of information about your assets as it is controlled by the device itself, not by potentially vulnerable computer software.

However, personal security devices aren’t intended to be a store of data - they’re a root of trust. Apps for these devices tend to be lightweight apps that work in conjunction with a host computer. The Secure Element places a limit on the storage capacity of these devices, so storing data encrypted on a host computer is preferred over storing data directly on the device. As such, the device contains a set of private keys that you can use to unlock your data and assets, without any risk of compromising the security of your assets in the event that your computer becomes infected with a virus. Applications that process large amounts of data may not be able to store all of the data on the device at once. There are two effective solutions to this problem: 1) stream data through the device while the application processes it (for example, it may be encrypting / decrypting the data) or 2) derive a secret on the device (preferably from the master seed) and use it on the computer to process the data (this is how the PGP app works).

This is an important concept: personal security devices are a secure portal to your assets, not a bank. If your device gets lost or destroyed, your assets are still safe. The key to this portal is your “master seed”. That brings us to our next section, *The Master Seed*. 
Applications running on Ledger personal security devices are designed to be lightweight and easily recoverable. Ledger achieves both of these goals by using **hierarchical deterministic key generation**. Hierarchical deterministic key generation is used by applications to derive a theoretically infinite number of cryptographic secrets from a single master seed. In this way, your cryptocurrency private keys, passwords, and other cryptographic secrets can all be determined and intrinsically “stored” in a single master seed. Because of this, the device’s apps don’t have to store their own private keys, because they can all be generated on-demand by the device from the master seed. This means that if your device is lost, destroyed, or reset then all you need is your master seed to recover your secrets. In addition, an application that supports this scheme can be deleted and reinstalled without losing any secure data or assets. Your master seed is randomly generated for you when you first set up your Ledger device, and then you just need to write it down to allow you to recover your device in the future.

### 3.1 Generation and Serialization

The master seed is the key to your cryptographic secrets. As such, the master seed should be something that is long and nearly impossible for anyone to guess. Your master seed isn’t like a password or a PIN (Ledger devices already have PINs to protect access to them), instead it’s the root of all of your cryptographic secrets. You only need to access it when you first write it down on paper when your device randomly generates it for the first time, or in the future if you have to recover your device.

Ledger uses a standard called **BIP 39** for the generation and interpretation of the master seed on all of our devices. BIP 39 is an industry standard used by many other hierarchical deterministic wallets. The exact type of BIP 39 seed used by Ledger devices by default is a 24-word mnemonic that consists of only the 2048 words from the **BIP 39 English wordlist**. Here’s how a BIP 39 24-word mnemonic seed is generated:

1. The device generates a sequence of 256 random bits using the true random number generator (TRNG) built into the device’s Secure Element.
2. The first 8 bits of the SHA-256 hash of the initial 256 bits is appended to the end, giving us 264 bits.
3. All 264 bits are split into 24 groups of 11 bits.
4. Each group of 11 bits is interpreted as a number in the range 0 - 2047, which serves as an index to the BIP 39 wordlist, giving us 24 words.
The result of this process is that your device will generate a single mnemonic seed out of $2^{256}$ possible mnemonic seeds (That's one of 115 792 089 237 316 195 423 570 985 008 687 907 853 269 984 665 640 564 039 457 584 007 913 129 639 936 possible mnemonic seeds). Note that while the first 23 words are completely random, the final word was derived from 3 random bits and 8 calculated bits from the SHA-256 hash. This means that the final word can act like a checksum - if you input an incorrect seed into the device while recovering it, it is possible for the device to detect that the inputted seed is invalid.

**Note:** Can someone guess my seed? There are $2^{256}$ different possible 24-word mnemonic seeds. For comparison, the number of atoms on Earth is estimated to be around $2^{166}$. Just let that sink in for a second. The chance of someone else being able to guess your seed is astronomically small, to say the least.

But all we have here is a sequence of 24 words, how can the device use this as a cryptographic secret?

After the 24-word mnemonic is generated, it has to be converted into a binary seed by the device (this process is called serialization). This is done using the PBKDF2-HMAC-SHA512 key derivation function to convert your mnemonic seed and an optional passphrase of your choosing into a 512 bit binary seed. This BIP 39 passphrase can be set on your device through the Settings menu, and it can be changed at will without resetting your device (in fact, you can have multiple passphrases loaded onto the device at once if you wish). By changing the passphrase, the resulting 512 bit binary seed is completely changed.

This 512 bit binary seed is the root of your device’s cryptographic secrets. Every cryptographic secret that your device needs (cryptocurrency private keys & addresses, passwords, etc.) can be derived from this 512 bit binary seed. We’ll explore how an infinite number of cryptographic secrets can be derived from this one seed in the next section, *HD Key Generation*. 
In the previous section we discussed how Ledger devices use a master 24-word mnemonic seed to derive a theoretically infinite number of cryptographic secrets. Though it might seem impossible at first glance, this can be done using nothing more than some mathematical sorcery. The process used to do this is called hierarchical deterministic (HD) key generation.

The process for HD key generation used by all Ledger devices (and many other HD wallets) is defined by BIP 32. Here’s how it works.

4.1 The Master Node

Hierarchical deterministic key generation involves creating a theoretically infinite tree of cryptographic secrets. The root of this tree from which everything is generated is called the master node. The master node is derived directly from the master binary seed described in the previous section using HMAC-SHA512. The master node of your wallet is all the information you need to access an infinite hierarchy of private keys. As such, you should take good care to keep your mnemonic seed safe.

4.1.1 Passphrases

Passphrases are a standard feature of BIP 39 supported by many HD wallets. Passphrases are an optional way of adding additional data to the master seed before deriving the master node of the HD wallet. By specifying a different passphrase (or no passphrase at all), the value of the master node is completely changed. As you will discover as you read the rest of this section, changing the master node of the HD wallet completely changes all of the information derived from it (for example, your Bitcoin addresses). In this way, you can keep the same master seed on a device, but use different passphrases to access completely unique, separated wallets.
4.2 An Infinite Tree

Descending from the master node is a tree of an infinite number of private keys. All you need to determine a particular key is the master node (your mnemonic seed) and the location of the key in the tree (called its “path”). This tree is created using an incredibly powerful algorithm defined by BIP 32 called the CKD function. All you need to know about this magical algorithm is that it can be used to calculate any node on the HD tree given the node’s position and the value of the master node. Essentially, the CKD function is applied to the master node a number of times in order to “scramble” its bits and output a brand new node. This algorithm is magical because it is impossible to reverse. Given a private key on your tree, it is impossible to go backwards up the tree and determine your master node.\footnote{Technically it isn’t impossible to determine a node given the child node and the corresponding index, but there is no known attack to do this faster than a properly executed brute-force attack. See this section of BIP 32 for details.}

4.3 Child Key Derivation Function

The HD tree is made up of “nodes”. Using the CKD function, many “child” nodes can be derived from a single “parent” node. Each node contains three pieces of information: a private key, a public key, and a chain code. In the case of a cryptocurrency wallet, the private key is the part that is used to sign transactions and the public key is used to generate the corresponding cryptocurrency address. The node’s chain code is an extra little bit to prevent someone from determining the children of a node using only the node’s public and private keys.

The CKD function has been designed in a way that provides great flexibility in the way child nodes are derived. Specifically, instead of requiring the public key, private key, and chain code to derive a child node, the CKD function can be used to derive child public keys from the parent public key and child private keys from the parent private key. Additionally, BIP 32 defines the concept of “hardened” child nodes. If a child node is “hardened”, then its public key cannot be determined from its parent node’s public key.

These are the transformations that the CKD function can perform:

- parent private key & chain code  child private key & chain code
- parent public key & chain code  child public key & chain code (unless the child node is hardened)
- parent private key & chain code  child public key & chain code

Note that it is impossible to derive a child private key from a parent public key. This model has some interesting consequences, mainly that the parent public key and chain code can be shared which allows someone to derive all child public keys (unless the child nodes are hardened), but no private keys.

For example, this model can be used to facilitate the process of auditing an HD wallet. By sharing the public key and chain code of all accounts in an HD wallet, one can give an auditor the ability to view all addresses in the wallet (without having the associated private keys). As such, the auditor could determine what money is going where, without having access to any private keys, meaning the auditor couldn’t sign any transactions. We’ll talk about what “accounts” are in the next section.

For a more detailed list and explanation of some of the use-cases made possible by this model, see this section of the BIP 32 specification.

In the next section, Applications of HD Trees, we’ll talk about how all of these features are used by HD wallets and other applications.

\textbf{Tip:} If you’d like to play with BIP 39 mnemonics or BIP 32 derivation on a computer, take a look at this tool: https://iancoleman.io/bip39/.
In the previous section, *HD Key Generation*, we described how an infinite number of cryptographic secrets can be generated from a single master seed. In this section, we’ll talk about how Ledger applies this concept to develop applications for our *personal security devices* that are lightweight and easily recoverable.

### 5.1 Coin Types

So how are HD trees useful? Simple: developers of different cryptocurrencies got together and reserved a space on the tree. Developers for these cryptocurrencies each specified which “location” on the tree they wanted users to store their private keys (called the coin’s “path”). This path is public information, because nobody can determine the actual value of your keys without knowing your master node (your mnemonic seed). The paths that correspond to various cryptocurrencies, and the paths used to derive cryptocurrency addresses and private keys, are defined by BIP 44 (which in turn defers coin type registration to SLIP 44).

For each cryptocurrency, there is a node on the tree where all of the keys for that coin begin (from this point on, we’ll call this the “coin type root node”, for lack of a better term). Since this is a hierarchical deterministic tree, we can apply the same rules to that coin type root node as we applied to the master node. Just as an infinite tree of keys descended from the master node, there is also an infinite tree of keys descending from the coin type root node. This means that if you have a 24-word mnemonic seed, even if you don’t use Bitcoin, you have a virtually infinite\(^1\) number of Bitcoin addresses. You have a virtually infinite number of Ethereum addresses, too.

**Warning:** In the Ledger Ethereum Wallet desktop app, addresses are not derived according to BIP 44. Instead of using the derivation path \(m / 44' / 60' / \text{account}' / \text{change} / \text{address\_index}\) (as defined by BIP 44 for Ethereum), the Ledger Ethereum Wallet desktop app uses the derivation path \(m / 44' / 60' / 0' / \text{address\_index}\).

---

\(^1\) In this case, virtually infinite = \(2^{63} = 9,223,372,036,854,775,808\).
5.2 How does my wallet know which addresses I've used?

In most BIP 44-compliant HD wallet programs (including the Ledger Bitcoin Wallet desktop app) your addresses are split into “accounts”. You can split your coins across multiple accounts in the same way you might with multiple bank accounts. You can have an account for savings, an account for donations, an account for common expenses, etc. Each of these accounts, there are two virtually infinite “chains” of addresses: an external chain and an internal chain (commonly called the “change address” chain). Each of the addresses in these chains are given numbers (called the “address index”) starting at 0.

When your wallet software starts up for the first time, it searches the blockchain for transactions involving address number 0 for each address chain. If it finds any transactions, then the address is displayed by the wallet software and the wallet searches for transactions involving address number 1. This process continues until the wallet finds an address that you haven’t used yet. It displays this address to you, then stops searching for more addresses in the chain, making the assumption that no more of them have been used yet.\(^2\)

In this way, all of the important information about your transaction history is stored in the blockchain, so none of it has to be stored in the hardware wallet itself. If your wallet is destroyed / reset, all you need to do is input your 24-word mnemonic seed into a new HD wallet and the wallet will search the public blockchain for all of the information it needs.

5.3 Summary

An HD wallet has a few very important properties, so let’s reiterate:

- The “tree” that makes up an HD wallet is generated using nothing more than your 24-word mnemonic seed. This is all you need to generate all of your cryptocurrency addresses, and sign transactions.
- No private keys need to be saved, anywhere. Even if you continue to receive Bitcoins to more and more of your addresses, you don’t need to save any of those private keys anywhere. This is because they were there all along, in the HD tree.
- The HD wallet standards used by Ledger (BIPs 32, 39, and 44) are industry standards. Your 24-word mnemonic seed will work with any other HD wallet that supports these standards, not just with Ledger wallets.

5.4 Going Beyond Cryptocurrencies

Hierarchical deterministic wallets are useful for a lot more than just securing assets on blockchains. You could also expand this concept and use your master seed to derive passwords, PGP keys, SSH keys, etc.

For example, Ledger is currently developing a Password Manager app. It is currently available for the Ledger Nano S in an early alpha form by clicking “show developer items” in the Ledger Manager.

The password manager app works by converting a node on the HD tree into a text password that looks like random gibberish. When you create a new password, you must enter a label for the password (for example: “Google”, “Dropbox”, “Twitter”, etc.). **The location of a password's node on the HD tree is encoded in the corresponding password's label.** When you ask the Password Manager app for your password, it uses the password’s label to locate a specific node on the HD tree, and then it converts that node into a text password. You cannot specify your own passwords to use, the passwords are generated for you.

---

\(^2\) In this example, we described a wallet that operates with a gap limit of 1. Different wallet software use different values, but BIP 44 currently recommends a gap limit of 20.
Fig. 1: A diagram of a hierarchical deterministic wallet; everything in the box labelled “hierarchical deterministic tree” is recoverable if your wallet is lost, destroyed, or reset because it is all derived from the master seed.
So how do I know that the apps I’m running on my Ledger Nano S are doing what I intend them to do? What protects me from getting a virus on my Ledger Nano S?

Ledger solves these problems by utilizing the application isolation technology available in the Secure Element - the ARM Memory Protection Unit and Operating Modes. The Memory Protection Unit is used to natively isolate each app to its own memory region, and apps run in User mode whereas the operating system runs in Supervisor mode. By restricting the device to a single-task model where only one app can run at a time, and each app is isolated from the rest of the device, apps are prevented from accessing your cryptographic secrets on the device unless you explicitly give them permission to.

The access that applications have to cryptographic secrets managed by the operating system can be configured when loading an application onto the device. Instead of accessing secrets like the device’s master seed directly, applications instead have to request the operating system to derive a node from the master seed by providing the operating system with a path to the requested node. When the application is loaded, the BIP 32 paths that the application is permitted to derive nodes from are specified. If the application requests the operating system to derive a node on a path that it is not permitted to use, the request is denied. In this way, many different applications can be loaded onto the device, and each of them can be restricted to a specific subtree of the HD tree depending on the application’s purpose. This process of requesting the operating system to perform an operation as Supervisor is called a syscall, and we’ll discuss it further in later sections.

Still, the importance of only installing apps that you can trust should not be understated. In the next section, we’ll talk about how Ledger’s operating system provides attestation features that allow the device to verify the authenticity of the apps that you’re installing by checking a digital signature that is sent along with apps when they’re loaded onto the device. Additionally, we’ll discuss how Ledger’s platform is open-source friendly which enables you to review the apps that you’re using to manage your assets.
This chapter contains detailed documentation about the core features offered by BOLOS, and how they can be utilized by BOLOS applications and end-users. We’ll discuss how BOLOS manages the master device seed and the device private key, and how it can be used for attestation purposes. We’ll also describe the hardware architecture that is common between all BOLOS devices.
The operating system behind all Ledger personal security devices is called the Blockchain Open Ledger Operating System, or BOLOS for short. BOLOS provides a lightweight, open-source framework for developers to build source code portable applications that run in a secure environment. BOLOS is a way of turning hardware wallets into fully fledged personal security devices.

BOLOS allows users to review and install applications that let them do more with their cryptographic secrets, while protecting the device and other applications from malicious code. The key to BOLOS’s open-source friendliness and ability to limit the exposure of user’s cryptographic secrets to their apps is its application isolation technology.

BOLOS is organized into the following modules:

- An input / output module which allows applications executing in a secure environment to communicate with the outside world and third party peripherals
- A cryptography module that implements low level cryptographic primitives and provides access to hardware acceleration where available
- A persistent storage module that lets applications store data securely on the device
- A personalization module for interfacing with the device master seed
- An endorsement & application attestation module allowing BOLOS applications to provide proof of execution
- A user interface module for rendering the GUI and handling user input (eg. via buttons on the device)

8.1 The Dashboard

All BOLOS devices have a special app installed that runs on the OS with certain special privileges called the Dashboard application or the PSD Content Manager. The dashboard app contains the main GUI that the user sees when they aren’t in any other app. This is what the users use to enter their master seed, and its what they use to launch other applications. The dashboard application is also what the host computer communicates with when loading or deleting apps off of the device.

An important component of the dashboard is the BOLOS UX, which is the implementation of the device-wide user interface that all applications need to interface with for certain device-wide UI features (like screen locking). The UI
of the default dashboard app that is built into the firmware for the Nano S is also available as an external application that can be loaded onto the device on GitHub.
In this section, we’ll discuss some of the features that are built into BOLOS. These features are available through the *dashboard app* and / or can be utilized by userspace applications.

### 9.1 Management of Cryptographic Secrets

There are two important cryptographic secrets that are stored and managed by BOLOS that will be discussed in this section: the *Device* keypair (which is generated in-factory) and the *BIP 32 master node* (which is derived from the user’s BIP 39 mnemonic seed). Both of these secrets are stored by BOLOS and are not directly accessible to applications for security reasons. The Device keypair can be used indirectly by applications for purposes of *application attestation*. Applications can derive secrets from the BIP 32 master node using a system call to BOLOS, provided the app was given the appropriate permissions when loaded onto the device.

#### 9.1.1 Passphrases in BOLOS

Since firmware version 1.3 on the Ledger Nano S, BOLOS allows users to load multiple **BIP 39 passphrases** onto the device at once. As described in the *previous chapter*, passphrases are a method to add additional entropy to the BIP 39 master seed in order to completely change the *HD tree*. Users can set a temporary passphrase which is activated until the device is disconnected, or store a passphrase on the device by attaching it to a PIN. When a passphrase is attached to a PIN, it is only activated when the user unlocks the device using the PIN corresponding to that passphrase. See our Help Center article on the advanced passphrase options for more information about using passphrases.

When a passphrase is activated, the binary seed derived according to BIP 39 is changed and as such the entire HD tree is changed. This means that using a different passphrase causes applications that derive information from the HD tree (like cryptocurrency wallet applications) to derive entirely different information (different cryptocurrency addresses will be generated).
# 9.2 Attestation

Attestation is a process used by Ledger devices to prove that they are a genuine Ledger device, and not a knock-off or fake version. It can be used by BOLOS when connecting to a host computer to prove that the device has not been tampered with. It can also be used by applications to prove that they are running on a genuine Ledger device. BOLOS also supports endorsement of the device by third parties (called Owners) for attestation purposes.

## 9.2.1 Anti-Tampering with Attestation

![Image of non-genuine Ledger wallet]

This Ledger Wallet is not genuine

For safety reasons, you are not authorized to use this application.

Please visit our help center at support.ledgerwallet.com for more information.

Ledger devices are protected from interdiction attacks (being tampered with while en route from Ledger’s warehouses to your home) due to anti-tampering technology built into the firmware. Using attestation, the authenticity of the device is verified in software every time you plug it into one of the Ledger Chrome applications.

When all Ledger devices are provisioned in the factory, they first generate a unique Device public-private keypair. The Device’s public key is then signed by Ledger’s Issuer key to create an Issuer Certificate which is stored in the device. This certificate is a digital seal of authenticity of the Ledger device. By providing the Device’s public key and Issuer Certificate, the device can prove that it is a genuine Ledger device.

When the Ledger device connects to one of the Ledger Chrome applications, the device uses the Issuer Certificate to prove that it is an authentic device (this takes place during establishment of the Secure Channel, as we’ll discuss later in this section). If an attacker created a clone of the device running rogue firmware, this attestation process would fail and the device would be rejected as non-genuine. It is impossible for an attacker to replace the firmware on the device and have it pass attestation without having a Device private key and the corresponding Issuer Certificate, signed by Ledger.

It is incredibly unlikely for the Device private key to become compromised, because the Secure Element is designed to be a stronghold against such physical attacks. It is theoretically possible to extract the private key, but only with great expense and time, so only an organization such as the NSA could do it.

**Tip:** For more information about the benefits of Ledger’s use of a Secure Element for verifying device authenticity, see our blog post How to protect hardware wallets against tampering (though keep in mind that not all of the information in this article applies to Ledger’s latest products).
9.2.2 Endorsement & Application Attestation

As discussed in the previous subsection, the *Device* private key can be used to prove authenticity of a Ledger device. However, direct access to the device private key is limited to BOLOS, so it can’t be directly utilized by individual applications on the device (to avoid compromising the key). However, applications can indirectly use the Device private key for attestation purposes by generating attestation keypairs.

Attestation keypairs can be generated on demand by the user for applications to use. An attestation key can be setup using the *endorsementSetup.py* or *endorsementSetupLedger.py* Python loader scripts. When generating an attestation keypair, the host computer connects to the dashboard application and initiates a Secure Channel before instructing the device to create an attestation keypair. The device generates a new attestation keypair and signs it using the *Device* private key to create a Device Certificate. The device then returns the attestation public key, the Device Certificate, and the Issuer Certificate over the Secure Channel to the host. The host, which may be Ledger or a third party, then signs the attestation public key with an *Owner* private key, thus creating an Owner Certificate which is sent back over the Secure Channel and stored by the device (in this way, the Owner “endorses” the authenticity of the device). The device can then prove that the attestation key belongs to a genuine Ledger device using the Device Certificate and the Issuer Certificate, and that the attestation key is trusted by the Owner using the Owner Certificate.

The attestation keys are not accessible to apps directly, instead BOLOS provides attestation functionality to userspace applications through cryptographic primitives available as system calls. There are two different Endorsement Schemes available to applications (Endorsement Scheme #1 and Endorsement Scheme #2). When creating an attestation keypair, the user must choose which scheme the keypair shall belong to. Applications can then use that keypair by using the cryptographic primitives offered for the appropriate Endorsement Scheme.

Endorsement Scheme #1 offers two cryptographic primitives:

* `os_endorsement_key1_get_app_secret(...)`: Derive a secret from the attestation private key and the hash of the running application.

* `os_endorsement_key1_sign_data(...)`: Sign a message concatenated with the hash of the running application using the attestation private key (this signature can be verified using *verifyEndorsement1.py*).

Endorsement Scheme #2 offers a single cryptographic primitive:

* `os_endorsement_key2_derive_sign_data(...)`: Sign a message using a private key derived from the attestation private key and the hash of the running application (this signature can be verified using *verifyEndorsement2.py*).

For an example of how these features may be used, check out blue-app-otherdime and this blog post which discusses the app in detail.

9.2.3 Attestation Chain of Trust

This diagram shows the chain of trust of our attestation model. All data signed by the attestation keys can be trusted to have been signed by an authentic Ledger device. This is because the Device Certificate is proof that the attestation keys belong to a device, and the Issuer Certificate is proof that the device is genuine. Additionally, the Owner Certificate is proof that the attestation keys are trusted by Owner (which may be Ledger or a third party).

9.3 Secure Channel

Throughout the standard device lifecycle, it is possible for a host computer to establish a Secure Channel with a device to verify its authenticity and to securely exchange secrets with it.

As discussed in *Anti-Tampering with Attestation*, the authenticity of a Ledger device can be verified when it connects to a host computer by requesting the device’s Issuer Certificate, which is signed by Ledger. This is done when establishing a Secure Channel with the device. However, the Secure Channel is not only a means to verify the authenticity
of a Ledger device, it also allows the host computer to establish an encrypted communication channel with the device. Only the dashboard application is able to establish a Secure Channel with the host computer, as doing so requires access to the Device private key.

The Secure Channel protocol is built on top of the APDU protocol used to communicate with the device over USB. As such, the protocol consists of a series of Command APDUs being sent from the host computer, and then associated Response APDUs being sent back from the device, with a one-to-one correspondence. The Secure Channel exists between two parties: the Signer and the Device. The Signer is the remote host connecting to the device. This may be the Issuer (Ledger) connecting to the device through our APIs, a Custom Certificate Authority connecting to the device using a previously enrolled Custom CA public key, or another end-user using a randomly generated keypair.

When establishing the Secure Channel, both parties (the Signer and the Device) generate an ephemeral keypair which is later used to calculate a shared secret using ECDH for encrypted communications between the two parties. Both parties prove that they trust their respective ephemeral public keys by each providing a certificate chain. These certificate chains incorporate both a Signer nonce and a Device nonce to avoid reuse of the certificates by an eavesdropper. If the root certificate in the certificate chain provided by the Signer is signed by a party that is trusted by the device, then the device grants the remote host special permissions after establishing the Secure Channel. For example, if the root certificate in the Signer’s certificate chain is signed by a previously enrolled Custom CA keypair or Ledger’s Issuer keypair, then the host can add or remove apps from the device without the user’s confirmation.

The process of establishing a Secure Channel is illustrated in the following diagram.

In the above diagram, during segment (6), the Device provides a Signer serial. The Signer serial is a number stored by the device which identifies the specific Issuer keypair used to sign the device’s Issuer Certificate, as Ledger does not use the same Issuer keypair for every device.

The Signer certificate chain is generated, sent to the device, and verified from (7) to (11). The Device certificate chain is generated, sent to the Signer, and verified from (12) to (16). In this example, both certificate chains consist of two certificates. The root certificate in the Signer certificate chain is self-signed. The final certificate in the Signer certificate chain is signed by the Signer and verifies the authenticity of the Signer ephemeral public key. The root certificate in the Device certificate chain is the Issuer Certificate (as such, verifying this certificate implicitly verifies
Fig. 2: An admittedly not-so-simple diagram of the Secure Channel protocol handshake
the authenticity of the device). The final certificate in the Device certificate chain is signed by the Device and verifies the authenticity of the Device ephemeral public key.

### 9.4 Custom CA Public Key Enrollment

*Custom Certificate Authorities* have the option to generate a keypair (using `genCAPair.py`) and enroll their public key onto the device (using `setupCustomCA.py`). Enrolling the Custom CA public key onto the device gives them the following special privileges:

- The Custom CA can open authenticated *Secure Channels* with the device (using the `--rootPrivateKey` option of the Python loader scripts).
- The Custom CA can sign applications (using `signApp.py`) to create a signature which can be used to avoid the user confirmation when loading the app on the device.

This feature may be used by BOLOS application developers to simplify the development process, but it is intended to be much wider in scope than that. This feature may also be used by third party companies to give their own application manager permissions to manage the device without needing user confirmation on every action.

### 9.5 Parties Involved in our Model

Below is a definition of all of the parties involved in our public key cryptography model.

**Device**

*Device Certificate* The meaning of this term should be quite self-evident, however in our public key cryptography model it has a distinct meaning. Each Device has a *unique* public-private keypair that is known *only to that device*. In the factory, the Device generates it’s own public-private keypair. The Device’s private key is not known by Ledger. The Device public-private key pair can be used to sign certificates.

**Issuer**

*Issuer Certificate* The Issuer is the party that initially provisions the *Device*. This party is always Ledger. The Issuer has a public-private keypair that can be used to sign Issuer Certificates. Note that Ledger uses multiple Issuer keypairs, not just one.

**Owner**

*Owner Certificate* An Owner is simply a party that owns and / or verifies the authenticity of a Ledger device. An Owner has a public-private keypair that can be used to sign certificates. A single *Device* can have zero or more Owners, and the Owner doesn’t have to be Ledger. The device uses Owner Certificates exclusively for the purposes of *application attestation*.

**Custom CA**

*Custom CA Certificate* A Custom Certificate Authority has a public-private keypair, where the public key is *enrolled on the device*. The Custom CA’s private key can then be used to establish authenticated *Secure Channels* with the device and sign applications.

A Custom CA may be a BOLOS application developer or a third party company that would like to give their application manager special administration permissions with a BOLOS device.
Ledger devices have a very unique architecture in order to leverage the security of the Secure Element while still being able to interface with many different peripherals such as the screen, buttons, the host computer over USB, or Bluetooth & NFC in the case of the Ledger Blue. In order to accomplish this, we attached an additional STM32 microcontroller (“the MCU”) to the Secure Element (“the SE”) which acts as a “dumb router” between the Secure Element and the peripherals. The microcontroller doesn’t perform any application logic and it doesn’t store any of the cryptographic secrets used by BOLOS, it simply manages the peripherals and notifies the Secure Element whenever new data is ready to be received. BOLOS applications are executed entirely on the Secure Element. In this section, we’ll take a look at the hardware architecture to better embrace the hardware related constraints before analyzing their software implications.

10.1 Multiple Processors: Secure Element Proxy

BOLOS is split between two hardware chips, one being secure (the ST31 Secure Element), and the other having JTAG enabled and acting as a proxy (the STM32 MCU).

Furthermore, the Secure Element is also split into two parts: the firmware which is under NDA and is therefore closed-source, and the SDK & application-loaded code which is open source friendly. The BOLOS firmware is responsible for low-level I/O operations and implements the SE-MCU link (though the handling of the protocol between the SE and the MCU is done by the currently running app).

BOLOS relies on the collaboration of both chips to empower Secure Element applications. At first glance, and even at second and all following, the Secure Element is a very powerful piece of hardware but lacks inputs / outputs. In our architecture, we solved this problem by appending the MCU which is full of inputs / outputs so it can act as a proxy for the Secure Element to explore new horizons. In a sense, the MCU can be seen as a supercharged coprocessor of the Secure Element. Not considering security implications (which is beyond the scope of this section), and thanks to a simple asynchronous protocol, the Secure Element drives the proxy.

The SE-MCU link protocol is called SEPROXYHAL or SEPH in source code and documentation. The “HAL” stands for Hardware Abstraction Layer.
10.2 SEPROXYHAL

The SEPROXYHAL protocol is structured as a serialized list of three types of packets: Events, Commands, and Statuses. Since SEPROXYHAL is the only channel for the SE to communicate with the outside world, if there is an error at the protocol level (such as the order or formatting of Events / Commands / Statuses getting messed up), then the SE ends up completely isolated and unable to communicate. When developing an application this is typically the most common failure scenario. If this happens, the device must be rebooted to reset the SEPROXYHAL protocol state. Hopefully, multiple levels of software guards are implemented to avoid such cases.

The protocol works as follows:

1. The MCU sends an Event (button press, ticker, USB transfer, ...).
2. The SE responds with a list of zero or more Commands in response to the Event.
3. The SE sends a Status indicating that the Event is fully processed and waits for another Event.

As a matter of fact, due to buffer size, requests to display something to the screen are sent using a Status. When the MCU has finished processing the Display Status, it issues a Display Processed Event indicating that it is ready to receive another Display Status. As a result, displaying multiple elements on the screen (in order to build an entire user interface) must be done asynchronously from the core application logic. This process is facilitated by a UX helper implemented in the SDK, which will be discussed further in the next chapter.

The SE throws an exception to applications willing to send more than one Status in a row, without a new Event being fetched in between.
Fig. 2: SEPROXYHAL protocol concept
Due to its limited amount of RAM, the Secure Element is designed to only support one application running at a time. This isolated model implies that once the application is running, no other application can spuriously disturb the SE-MCU link. It also means that BOLOS can give the currently running application full control of I/O with the device’s peripherals. This model allowed the BOLOS architecture to be designed in a way that gives applications as much control over the device’s features as possible. In essence, each application runs in a “virtual” device and can reconfigure all of the hardware as it pleases. BOLOS isolates the application from the other applications on the device, and restricts access to all areas of flash memory other than those exclusively allocated for the running application.

This model has the tremendous advantage of not limiting what the application can do, however it also implies that every application has to do all of the heavy lifting involved in managing every layer of the transport protocols used to communicate with the world outside of the SE. Luckily, the SDK implements all I/O handling that typical applications need to do. However, developers have the option to customize I/O protocols for more specialized applications.

The above diagram shows a view of the system as seen by the application. The app directly accesses multiple peripherals and is the real brain of the device while it is running. Each box can be seen as a coprocessor, under direct command of the application.

Some peripherals not only receive commands from the SE, but also trigger events which are relayed back to the SE by the MCU. This is the case for buttons, activated upon user actions, and I/O peripherals which can perform background communication (for example, the USB controller) or convey requests to be processed by the application.

In this model, the application is at the center, and does not rely on any other embedded co-applications.

### 11.1 Delegation Model

Once BOLOS boots the application, BOLOS is not reachable anymore, it only provides basic services to the application during its execution via system calls. As a consequence, BOLOS does not process commands sent to the device from peripherals (like USB) and therefore BOLOS does not play a role in I/O handling.

Featuring these two key points, applications are in charge on the device. This allows them to customize not only the display, but user input actions, and by extension, the way the device is enumerated on USB. If an application requires Mass Storage emulation, or being seen as a WinUSB peripheral, it’s only a matter of event handling.
Fig. 1: The application-centric view of the BOLOS environment

Fig. 2: An overview of the USB delegation model
In this chapter we will provide a general tutorial for getting your BOLOS development environment set up, followed by some detailed explanations of the various components of the BOLOS SDKs and what userspace development entails. It is assumed that you have already read the BOLOS Platform chapter and are somewhat familiar with the BOLOS architecture.
Warning: Only Linux is supported as a development OS. For Windows and MacOS users, a Linux VM is recommended.

Developing and / or compiling BOLOS applications requires the SDK matching the appropriate device (the Nano S SDK or the Blue SDK) as well as the following two compilers:

- A standard ARM gcc to build the non-secure (STM32) firmware and link the secure (ST31) applications
- A standard ARM clang above 7.0.0 with ROPI support to build the secure (ST31) applications
- Download a prebuilt gcc from here

13.1 Setting up the Toolchain

The Makefiles used by our BOLOS applications look for gcc and clang installations using the PATH environment variable.

If you don’t want to install specific versions of clang and gcc directly on your system, simply prepend their location in your PATH environment variable.

```bash
# GCC
PATH=~/bolos-devenv/gcc-arm-none-eabi-5_3-2016q1/bin:$PATH

# Clang
PATH=~/bolos-devenv/clang+llvm-7.0.0-x86_64-linux-gnu-ubuntu-16.04/bin:$PATH
```

Cross compilation headers are required and provided within the gcc-multilib and g++-multilib packages. To install them on a debian system:

```
sudo apt install gcc-multilib g++-multilib
```
13.2 Setting up the SDK

Now that you have your toolchain set up, you need to download / clone the SDK for the appropriate Ledger device you’re working with. Make sure you checkout the tag matching your firmware version.

Ledger Nano S SDK: https://github.com/LedgerHQ/nanos-secure-sdk
Ledger Blue SDK: https://github.com/LedgerHQ/blue-secure-sdk

Finally, link the environment variable BOLOS_SDK to the SDK you downloaded. When using the Makefile for our BOLOS apps, the Makefile will use the contents of the SDK to determine your target device ID (Ledger Nano S or Ledger Blue). Even if you aren’t building an app, loading an app with the Makefile still requires you to have the SDK for the appropriate device linked to by BOLOS_SDK.

13.3 Python Loader

If you intend to communicate with an actual Ledger device from a host computer at all, you will need the Python loader installed. For more information on installing and using the Python loader, see BOLOS Python Loader. The Makefiles for most of our apps interface with the Python loader directly, so if you only need to load / delete apps then you don’t need to know how to use the various scripts provided by the Python loader, but you’ll still need it installed.

13.4 Building and Loading Apps

In this section, we’ll walk you through compiling and loading your first BOLOS app onto your device. Applications that support multiple BOLOS devices are typically contained within a single repository, so you can use the same repository to build an app for different Ledger devices. Just make sure that you’ve set BOLOS_SDK to the appropriate SDK for the device you’re using. The Makefiles used by our apps use the contents of the SDK to determine which device you’re using.

Firstly, download the boilerplate app.

```
git clone https://github.com/LedgerHQ/ledger-app-boilerplate.git
```

Now you can let the Makefile do all the work. The load target will build the app if necessary and load it onto your device over USB.

```
cd ledger-app-boilerplate/
make load
```

And you’re done! After confirming the installation on your device, you should see an app named “Boilerplate”. The app can be deleted like so:

```
make delete
```

The Sia app is a very well documented app from community. If you want to study a full fledged app, this is the one you should read:

```
git clone https://github.com/LedgerHQ/ledger-app-sia.git
```
Since BOLOS is designed based on a single-task model where only a single app runs at any given time, an application is independently responsible for a lot of the things a typical OS would do. These things include managing hardware like the device screen, buttons, timer, etc. as well as handling all I/O with peripherals. However, there are many instances where a BOLOS application has to request the operating system to perform a certain operation. This is done using a syscall.

When an application performs a syscall, the Secure Element switches to Supervisor mode and the OS performs the requested task before returning control back to the application, in User mode. All syscalls have a wrapper function in the SDK that can be used to invoke them. A syscall may be used to access hardware accelerated cryptographic primitives (most of these functions are defined in `include/cx.h` in the SDKs), to perform low-level I/O operations (like receiving / transmitting to the MCU), or to access cryptographic secrets managed by BOLOS (for example, to derive a node from the BIP 32 master node).

### 14.1 Error Model

If you are familiar with C programming, you will be used to error codes as the default error model. However, when programming in the embedded world, this traditional model reaches its limits, and can quickly overcomplicate large codebases. Therefore, we've implemented a try / catch system that supports nesting (direct or transitive) using the `setjmp` and `longjmp` API to facilitate writing robust code.

Here is an example of a typical try / catch / finally construct:

```c
BEGIN_TRY {
    TRY {
        // Perform some operation that may throw an error using THROW(...)  
    } CATCH_OTHER(e) {
        // Handle error  
    } FINALLY {
        // Always executed before continuing control flow  
    }
} END_TRY;
```
However there is a single constraint to be aware of with our try / catch system: a TRY clause must always be closed in the appropriate way. This means that if using a return, break, continue or goto statement that jumps out of the TRY clause you MUST manually close it, or it could lead to a crash of the application in a later THROW. A TRY clause can be manually closed using CLOSE_TRY. Using CLOSE_TRY is only necessary when jumping out of a TRY clause, jumping out of a CATCH or FINALLY clause is allowed (but still, be careful you’re not in a CATCH nested in a TRY).

You should use the error codes defined in the SDKs wherever possible (see EXCEPTION, INVALID_PARAMETER, etc. in os.h). If you decide to use custom error codes, never use an error code of 0.

Developers should avoid creating a new try context wherever possible in order to reduce code size and stack usage. Preferably, an application should only have a single top-level try context at the application entry point (in main()).

### 14.2 Syscall Requirements

BOLOS is based on an exception model for error reporting, therefore, it expects the application to call the BOLOS API using this mechanism. If an API function is called from outside a TRY context, then the BOLOS call is denied.

Here is a valid way to call a system entry point:

```c
BEGIN_TRY {
    TRY {
        cx_hash_sha512(...);
    } FINALLY {}  
} END_TRY;
```

However, as mentioned above, it is preferred to use as few try contexts as possible (not one per syscall). A single, top-level try context can be used to catch any exception thrown by any syscall performed by the application.
CHAPTER 15

Application Structure and I/O

Many of the existing BOLOS applications are based on a smartcard architecture. This is because BOLOS applications are not meant to run standalone, but rather assist a host process (on a computer / smartphone) to perform a secure task (signing a message, encryption / decryption, etc.). Therefore the device is commonly addressed using a command / response scheme. Numerous design decisions have been made when developing the SDKs in order to support this model.

However, the Event / Commands / Status model is designed to avoid limitations on the application, as it does not follow the command / response synchronous model. Developers are free to work around the model and redesign a custom event processing loop to suit their needs.

15.1 APDU Interpretation Loop

The command / response scheme used to address the device is similar to the ISO/IEC 7816-4 smartcard protocol. Each command / response packet is called an APDU. The application is driven by a never-ending APDU interpretation loop called straight from the application main function.

Each cycle of the APDU interpretation loop calls `io_exchange(...)` from the SDK, which first sends a response APDU (unless it’s the first call, in which case it sends nothing), and then receives the next command APDU.

However, sometimes, a user confirmation to perform a security action must be performed before replying to an APDU (for example, when signing a message). Such behavior is handled by replying no data to the command in the APDU interpreter by using the `IO_ASYNCH_REPLY` flag, then following the user action calling `io_exchange` with the `IO_RETURN_AFTER_TX` flag and with the amount of data to reply to the stalled command.

For an example of this feature, refer to `blue-app-samplesign`. In this app, when the command APDU to sign a message is received (line 510), the flag `IO_ASYNCH_REPLY` is set and no response APDU is sent. If the user approves the action, then the button push handler calls `io_seproxyhal_touch_approve(...)` which sends the response APDU using another call to `io_exchange(...)` with the `IO_RETURN_AFTER_TX` flag set (line 434). The same occurs if the user denies the action, in which case `io_seproxyhal_touch_deny(...)` is triggered.
15.2 Protocols

It’s important to understand that the APDU protocol used by most BOLOS applications is not implemented by BOLOS itself. Instead, the APDU interpretation is performed entirely by the SDK. This means that applications can choose to implement another protocol on top of the transport layer (USB HID, USB CCID, BLE, ...) instead of APDU. In fact, the same is true for the transport layer protocols. Applications can customize the way the application is enumerated as a USB device by the host.

Userspace applications are free to implement their own protocols on top of APDU

Userspace applications may interface with USB & BLE devices directly

APDU

USB HID  USB CCID  BLE

BAGL Elements (via a Display Status)

Other protocols interfacing with device hardware

Event / Commands / Status Protocol (SEPROXYHAL)

Physical SE-MCU Link

Fig. 1: Common protocols across BOLOS applications

15.3 Unprocessed Events

APDU processing relies on the BOLOS way of framing / transporting APDU packets. All event processing related to transfer operations (including USB) is performed within io_exchange (...). Not considering customization of the transport, some Events are not automagically processed by io_exchange(...) (for example: Button Push Events, Display Processed Events, Ticker Events, ...). In order to handle these events that cannot be handled automagically, io_exchange(...) calls io_event(...) which is defined by the application (not by the SDK).

Developers must take great care in the way they process Events. Events might occur in the middle of APDU transport (most likely Button Push or Ticker Events). As such, io_event(...) must return 1 if events are expected, otherwise the current APDU transport will be terminated (this feature could be used to implement a timeout, for example).

Thanks to a hardware buffer in the SE, it is impossible to miss an Event packet. And, due to the E/Cs/S protocol design, no Event will be sent by the MCU until a new Status is sent by the application.
CHAPTER 16

Display Management

The BOLOS SDKs contain a toolkit for building GUIs called the BOLOS Application Graphics Library (BAGL). BAGL defines a few useful types, most notably bagl_element_t. This type defines a single display element, such as a rectangle, a line of text, a touchable button (in the case of the Ledger Blue), et cetera. Therefore, an entire GUI consists of an entire array of such elements. As the device hardware, including the screen, is managed by the MCU, the BAGL elements need to be transported to the MCU over SEPROXYHAL in order to display them. This is done using a Display Status.

A Display Status may be used to send a single BAGL element to the MCU. However, due to the design of the E/Cs/S protocol, in order to send a sequence of BAGL elements the application must asynchronously send Display Statuses and wait for Display Processed Events before sending the next one, as well as handling the other Events that are received from the MCU in the mean time. As this is something that must be done by every BOLOS application, a set of utilities have been defined in the SDK to facilitate this process. These utilities also simplify the process of handling user input events, such as button presses.

16.1 Asynchronous Display and Interaction Helpers

To facilitate the process of implementing an asynchronous display manager loop, a set of macros have been defined in the BOLOS SDKs. All these macros have the prefix UX_, and use a global variable of type ux_state_t called ux to maintain the user interface state.

UX_INIT(...) This macro is to be called when initializing the application, prior to sending the first Display Status.

UX_DISPLAY(...) This macro takes an array of BAGL elements to be displayed and renders them, asynchronously. It is to be called when a new screen is to be displayed over the current one. This macro only sends the first element of the given array using a Display Status. Therefore, further Commands and Statuses are prohibited until the Display Processed Event is sent by the MCU.

UX_REDISPLAY() This macro restarts the process of drawing the current screen. It behaves like UX_DISPLAY(...), but takes no arguments.

UX_DISPLAYED() This macro returns 0 when the array requested to be displayed by UX_DISPLAY(...) or UX_REDISPLAY() has not been displayed entirely, or a non-zero value when it has.
UX_DISPLAYED_EVENT(...) This macro is to be called to handle Display Processed Events (generally in the
io_event(...) function). It displays the next element in the array given as a parameter to UX_DISPLAY(...). This macro sends a Display Status if an element remains to be displayed in the given array. Therefore, further Commands and Statuses are prohibited until the Display Processed Event is sent by the MCU.

UX_BUTTON_PUSH_EVENT(...) This macro facilitates handling of Button Push Events, by setting the button released flag and calling the button handler implicitly passed to UX_DISPLAY(...).

See the sample apps for examples of how to use these macros. The main concept to remember is that after a Display Status has been sent, the application must wait, asynchronously, for the Display Processed Event before being able to continue to display more elements of the GUI.

16.2 BOLOS UX

The BOLOS UX is the implementation of the device-wide user interface; it is a component of the dashboard. Applications delegate certain jobs to the BOLOS UX in order to retain consistency across all apps for certain UI components (like the status bar on the Ledger Blue), as well as to allow the operating system to override the application’s UI when necessary (for example, when locking the screen). The application interfaces with the BOLOS UX using os_ux(...), which is a syscall. However, applications don’t need to call this function directly as it is automatically called by the display interaction helpers (the UX_ macros).

Applications should delegate Events like Button Push Events to the BOLOS UX (in this case, using UX_BUTTON_PUSH_EVENT(...)) instead of handling them directly in case the BOLOS UX needs to override the application’s UI. If the event is consumed by the BOLOS UX (for example, a button press while the user is unlocking the screen) then the event is not passed on to the application.
BOLOS applications have access to two different types of memory in the Secure Element: a small amount of RAM for the call stack and certain global variables, and a considerably larger amount of flash memory for persistent storage. Access to flash memory is regulated by the Memory Protection Unit which is configured by BOLOS to prevent applications from tampering with parts of flash memory that they shouldn’t. However, applications are able to access the part of flash memory where their constant data and code is defined. This data includes code and \texttt{const} variables, but applications may also allocate extra space in NVRAM to be used at runtime for persistent storage.

### 17.1 Types of Memory

All global variables that are declared as \texttt{const} are stored in read-only flash memory, right next to code. All normal global variables that are declared as non-\texttt{const} are stored in RAM. However, thanks to the link script (\texttt{script. ld}) in the SDK, global variables that are declared as non-\texttt{const} and are given the prefix \texttt{N} are placed in a special write-permitted location of NVRAM. This data can be read in the same way that regular global variables are read. However, writing to NVRAM variables must be done using the \texttt{nvm_write(...)} function defined by the SDK, which performs a syscall. When loading the app, NVRAM variables are initialized with data specified in the app’s hex file (this is usually just zero bytes).

\textbf{Warning}: Initializers of global non-\texttt{const} variables (including NVRAM variables) are ignored. As such, this data must be initialized by application code.

### 17.2 Flash Memory Endurance

The flash memory for the \texttt{ST31G480}, which is the Secure Element used in the Ledger Blue, is rated for 500 000 erase / write cycles. This should be more than enough to last the expected lifetime of the device, but only if applications use it properly. Applications should avoid erasures as much as possible. Here are some techniques for avoiding wearing out the device’s flash memory.
Firstly, if you intend to be changing data in flash memory many times while an application is running, consider caching the data in RAM and then flushing to flash memory when the application has finished its operation. This of course has the downside of possible data loss if the user powers off the device (perhaps by unplugging it, in the case of the Nano S) before the data has been written to persistent storage. Secondly, developers should be aware that flash memory pages are aligned to 64-byte boundaries. The rating of 500,000 erase/write cycles mentioned earlier means that each page in flash memory is expected to survive 500,000 erasures. As such, one can develop an application that writes to as few pages as possible. For example, if you intend to store 32 bytes of data in flash memory, write amplification can be avoided by making sure that 32 bytes of data is contained entirely within a single page (and modified using only a single call to `nvm_write(...)`). If the data crossed a 64-byte page boundary, then writing to it once may require two pages to be erased instead of just one.

In the future, Ledger will provide various persistent storage utilities within BOLOS and the SDKs to simplify the process of using flash memory efficiently.

### 17.3 PIC and Model Implications

PIC stands for Position-Independent Code. The BOLOS toolchain produces PIC to allow for the code Link address to be different from the code Execution address. For example, the `main` function is linked in the generated application at address 0xC0D00000. However, the slot used when loaded into the Secure Element could be 0x10E40400. Therefore, if the code makes a reference to 0xC0D00000, even with an offset, it would be denied access as the application is locked by the Memory Protection Unit (not to mention, this is not the correct address of the `main` function at runtime).

The PIC assembly generator makes sure every dereference is relative to the Program Counter, and never to an arbitrary address resolved during the link stage. This behavior is supported by clang versions 4.0.0 and later.

Traditionally, PIC code implies the BSS segment (RAM variables) is at a constant offset of the code. For example, if code is at 0xC0D00000, then global vars may be at 0xC2D00000, so if loaded at 0x10E00000 then global vars would be at 0x12E00000. However, BOLOS uses a fixed address for global vars. The global variables start address and length are defined in the link script. Only the code is meant to be placed at different addresses (in flash memory, rather than RAM).

The model we chose has limitations, which are related to the way `const` data and code is referenced in other `const` data. Here is a simple example:

```c
const char array1[] = {1, 2, 3, 4};
const char array2[] = {1, 2, 3, 4};
const char *array_2d[] = {array1, array2};

void main() {
    int sum, i, j;
    sum = 0;
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 4; j++) {
            sum += array_2d[i][j]; // Segmentation Fault!
        }
    }
}
```

In the example above, when dereferencing `array_2d`, the compiler uses a link-time address (in the 0xC0D00000 space, following the previous examples). This is not where the program is loaded in memory at runtime. Therefore, when the dereference is executed, it causes a segmentation fault that effectively stalls the SE. Luckily, the solution is pretty simple, thanks to a small piece of assembly provided with the SDKs which is invoked with the `PIC(...)` macro. `PIC(...)` uses the current load address to adjust the link-time address in order to acquire the correct runtime address of `const` data and code. The above examples can be corrected by modifying the line where `array_2d` is dereferenced to do the following:
The same mechanism must be applied when storing function pointers in `const` data. The PIC call cast is just different. Additionally, if a non-link-time address is passed to `PIC(...)`, then it will be preserved. This is possible due to the wisely chosen link-time address which is beyond both real RAM and loadable addresses. For example, `PIC(...)` is used during a call to `io_seproxyhal_display_default(...)`, all display elements can hold a reference to a string to be displayed with the element, and the string could be in RAM or code, and therefore `PIC(...)` is applied to acquire the correct runtime address of the string, even if it’s in RAM.
The Secure Elements on top of which the BOLOS Operating System and the associated applications run imply a 32-bit alignment. This paragraph aims at explaining the C associated development constraints.

18.1 Alignment concept

The memory alignment is a concept which applies to memory and pointers:
- A memory address is ‘b-bits aligned’ when it is a multiple of b/8, b/8 being a power of 2,
- A memory address is said to be aligned when the data referenced by said address is b bits long, and said address is b-bits aligned,
- A pointer is ‘aligned’ when it points on aligned memory,
- A pointer is ‘unaligned’ when it points on unaligned memory.

18.2 Alignment constraints for basic types and structures

Implementing C source code with types and structures is not functionally impacted by the 32-bit alignment, except for potentially wasting a few bytes without even noticing.

It might be important to be aware of this paragraph contents when it comes to writing memory-efficient structures, once the application is compiled and loaded onto a device.

Within any application source code, the alignment of basic types will be considered as follows, at compilation time:
- char/unsigned char/int8_t/uint8_t: 8-bit aligned,
- short/unsigned short/int16_t/uint16_t: 16-bit aligned,
- int/unsigned int/int32_t/uint32_t: 32-bit aligned,
- any pointer: 32-bit aligned.
Please note that 8-bit aligned means that there is actually no alignment constraint.

The compiler will add padding in any structure which is not aligned by design, in order to respect:

- The alignment of each field associated to their respective length,
- The alignment of the whole structure, which shall have a total length, padding included, multiple of the largest field’s length.

For instance the following structure is 8 bytes long before compilation:

```c
// Before compilation
struct Example1
{
    char    field_1;
    short   field_2;
    int     field_3;
    char    field_4;
};
```

However during compilation, the structure is modified to ensure the alignment, and will thus be 12 bytes long:

```c
// After compilation
struct Example1
{
    char    field_1;
    // Padding added for field_2 to start on a 16-bit aligned address
    char    padding_1;
    short   field_2;
    // With padding_1 being added, field_3 will start on a 32-bit aligned
    // address and no padding is required here.
    int     field_3;
    char    field_4;
    // The structure is aligned to the number of bits corresponding to the
    // largest field's alignment, in this case, due to field_3, 32-bits.
    // For the structure to be 32-bit aligned, it needs 3 more bytes of padding.
    char    padding_2[3];
};
```

In this example, it is possible to reorganize the structure’s fields to avoid alignment-induced padding, but sometimes padding will not be avoidable.

One can order the structure fields according to their length in decreasing order:

```c
// Before compilation
struct Example1_reordered
{
    int     field_3;
    short   field_2;
    char    field_1;
    char    field_4;
};
```

```c
// After compilation
struct Example1_reordered
{
    int     field_3;
    // No need for padding since field_2 is already on a 16-bit aligned address.
    short   field_2;
};
```
One can also order the structure fields to make sure the minimum amount of padding bytes will be added by the compilation phase:

```c
// Before compilation
struct Example1_reordered_other_way {
    int field_3;
    char field_1;
    char field_4;
    short field_2;
};

// After compilation
struct Example1_reordered_other_way {
    int field_3;
    // No need for padding for char types.
    char field_1;
    char field_4;
    // No need for padding since field_2 is already on a 16-bit aligned address.
    short field_2;
    // No need for padding since the structure is 8 bytes long and thus, its length
    // is already a multiple of its largest field's length.
};
```

### 18.3 Alignment constraints for pointers

Using pointers within C source code might be functionally impacted by the 32-bit alignment in a specific case: when the pointer points on a memory area which type differs from the pointer, and is dereferenced.

Dereferencing unaligned pointers within an application stalls the device.

Usually, pointers are used to store the address of an element which type corresponds to the pointer one, and for simple example:

```c
uint16_t *pointer;
uint16_t array[10];

// Pointer positioning is perfectly fine.
pointer = &array[3];

// Dereferencing this pointer is also perfectly fine, since the pointed memory is aligned in accordance with the pointer type.
*pointer = 0x0001;
```

However, if we use a pointer with a specific type to store the address of a memory area declared with another type (usually with an alignment-related size less than the pointer one), it can lead to hardware faults and stall the device:
uint16_t *pointer;
uint8_t array[10];

// Case where it will work even if not advised.
// Pointer positioning is fine.
pointer = (uint16_t*)array[2];

// Dereferencing this pointer is also fine: the pointed memory is aligned
// in accordance with the pointer type (because the offset 2 in the array variable
// is a multiple of 16 bits).
if (*pointer == 0x0001) {
    do_something();
}

// Case where it will stall the device.
// Pointer positioning is fine, but it is unaligned.
pointer = (uint16_t*)array[3];

// Dereferencing this pointer will stall the device: the pointed memory is not aligned
// in accordance with the pointer type (because the offset 3 in the array variable
// is not a multiple of 16 bits).
if (*pointer == 0x0001) { /* This dereferencing stalls the device. */
    do_something();
}

The same reasoning applies to pointing on structures:

// Same example as within the previous paragraph, being ordered
// makes it 8 bytes long.
struct Example1_reordered {
    int field_3;
    short field_2;
    char field_1;
    char field_4;
};

Example1_reordered *pointer;
uint8_t array[32];

// Case where it will work even if not advised.
// Pointer positioning is fine.
pointer = (Example1_reordered*)array[8];

// Dereferencing this pointer is also fine: the pointed memory is aligned
// in accordance with the pointer type (because the offset 8 in the array variable
// is a multiple of the structure's size after compilation).
if (pointer->field_2 == 0x0001) {
    do_something();
}

// Case where it will stall the device.
// Pointer positioning is fine, but it is unaligned.

pointer = (Example1_reordered*)array[3];

// Dereferencing this pointer will stall the device: the pointed memory is not aligned
// in accordance with the pointer type (because the offset 3 in the array variable
// is not a multiple of the structure’s size after compilation).
if (pointer->field_2 == 0x0001) { /* This dereferencing stalls the device. */
    do_something();
}

Unaligned pointers can thus occur in cases where a pointer:

- declared as positioning on some data type (or structure)
- is used to point on a memory area actually containing another type of data,
- and is dereferenced.

In order to produce C source code robust to alignment constraints, one need to avoid using pointers in such a way.

### 18.4 External links

In this section, we’ll walk you through a lot of concepts that are hard to grasp when developing on the BOLOS platform, and we’ll provide some analysis of common failure scenarios that you might experience while developing applications.

19.1 Not Enough RAM

At the time of this writing, the default link script provided by the SDK for the Ledger Nano S allocates 4 KiB of RAM for applications to use. This 4 KiB has to be enough to store all global non-const and non-NVRAM variables as well as the call stack (which is currently set to 768 bytes by default, also defined in the link script).

This is the linker error you will experience if you declare too many global non-const and non-NVRAM variables to fit in RAM:

```
bin/app.elf section `.bss' will not fit in region `SRAM'
```

The only solution to this problem is, of course, using less RAM. You can accomplish this by making your application’s memory layout more efficient. Alternatively, if you’re feeling adventurous, you can attempt to modify the link script (script.ld in the SDKs) to optimize the space allocated for the call stack. If you choose to pursue the latter option, we recommend you read the next section as well.

19.2 Stack Overflows

Determining the exact amount of the call stack used by your application can be difficult to do without simply running your application. The technique we recommend for avoiding stack overflows is using a stack canary. Creating a stack canary involves setting a magic value at the start of the stack area (the stack grows towards lower addresses, so a canary at the start of this region will be located at the top of the stack), and then the canary is checked regularly. If the canary was modified, then this means there was a stack overflow.

In a future version of the BOLOS SDKs, this feature will be implemented automatically. Until then, this is the recommended way to implement a stack canary:
The canary should be checked regularly. For example, you could run the check every time \texttt{io_event(...) \ldots} is called.

### 19.3 Error Handling

Error handling in C can sometimes be a bit counter-intuitive. With our error model, there are two common failure scenarios.

Firstly, you must take care to always close a try context when jumping out of it. For example, in the block of code below, the \texttt{CLOSETRY} macro must be used to close the try context before returning from the function in the case that \texttt{i > 0}. However, in the \texttt{CATCH} clause, the try has already been closed automatically so \texttt{CLOSETRY} is not necessary.

```c
bool is_positive(int8_t i) {
    BEGIN_TRY {
        TRY {
            if (i == 0)
                THROW(EXCEPTION);
            if (i > 0) {
                CLOSE_TRY;
                return true;
            }
        } CATCH_OTHER(e) {
            return false;
        } FINALLY {}
    } END_TRY;
    return false;
}
```

Another common failure scenario is caused by the compiler making invalid assumptions when optimizing your code because it doesn’t understand how our exception model works. To avoid this problem, when modifying variables within a try / catch / finally context, always declare those variables \texttt{volatile}.

```c
uint16_t multiply(uint8_t a, uint8_t b) {
    volatile uint16_t product = 0;
    volatile uint8_t multiplier = b;
    while (true) {
        BEGIN_TRY {
            TRY {
                if (multiplier == 0)
                    THROW(EXCEPTION);
                if (i < 0) {
                    return false;
                }
            } CATCH_OTHER(e) {
                return false;
            } FINALLY {}
        } END_TRY;
        return false;
    }
```
In the above example, \texttt{a} does not need to be declared \texttt{volatile} because it is never modified.

On another note, you should use the error codes defined in the SDKs wherever possible (see \texttt{EXCEPTION, INVALID_PARAMETER}, etc. in \texttt{os.h}). If you decide to use custom error codes, never use an error code of \texttt{0}.

### 19.4 Application Stalled

An application stalling when run on the device (the device’s screen freezes and stops responding to APDU) could be caused by a number of issues from the SE being isolated due to invalid handling of SEPROXYHAL packets, to a core fault on the device (perhaps due to a misaligned memory access or an attempt to access restricted memory). If this occurs, it is best to attempt to simplify the app and strip away as much code as possible until the problem can be isolated.

### 19.5 Unaligned RAM access

```c
uint16_t *ptr16 = &tmp_ctx.signing_context.buffer[processed];
PRINTF("uint16_t: %d", ptr16[0]);
```

\texttt{ptr16[0]} access can be stalling the app, even though \texttt{tmp_ctx.signing_context.buffer[processed]} (\texttt{unsigned char*}) can be accessed alright. This happens when pointer isn’t word-aligned, but word is access in RAM. Workaround is copying buffer into another location which is properly aligned (e.g. using \texttt{os_memmove}).

Please refer to the \texttt{alignment} page for further information.
Note: Ledger is currently working on more solutions to improve the coding experience on its products. In particular, a development board and an emulator are in the pipe, but there is not yet an estimated time of arrival for these.

Warning: The BOLOS development environment is required for the following article. It applies only for the Nano S, with its SE firmware either in version 1.5.5 or 1.6.0.

It is possible to install a debugging firmware on the device’s MCU that will enable printing text outputs from the device to a terminal. To do so, follow these steps:

1. First, download the updater and the debug firmware.
2. Exit any instance of Ledger Live, Ledger Chrome App, or any other program able to communicate with a Ledger device.
3. Now, plug your Nano S to your computer while keeping the left button pressed. Keep it pressed until the screen displays BOOTLOADER.
4. Fire a terminal and move to the directory containing the files downloaded at step 1.
5. Install the updater (only if you MCU firmware is not already in version 1.11, otherwise just go to step 6):

   ```
   python -m ledgerblue.loadMCU --targetId 0x01000001 --fileName blup_0.11_misc_m1.hex --nocrc
   ```

   Wait until BOOTLOADER is displayed again on the device’s screen.

   6. Install the debug firmware:

   ```
   python -m ledgerblue.loadMCU --targetId 0x01000001 --fileName mcu_1.11-printf_over_0.11.hex --reverse --nocrc
   ```

   If you can notice a small dbg block at the bottom of the screen, then it’s a success!
Fig. 1: A Nano S with special debug firmware

Uninstalling this special firmware is also very easy, first you need to download the normal firmware, then you can repeat the installation steps 2 to 5.

Finally, flash the normal firmware with this command:

```
python -m ledgerblue.loadMCU --targetId 0x01000001 --fileName mcu_1.11_over_0.11.hex -r --nocrc
```

The dbg block should now be gone.

### 20.1 PRINTF macro

The debug firmware enables the PRINTF macro, however you have to define it in your app’s Makefile. To do so, add this line in your Makefile:

```
DEFINES += HAVE_SPRINTF HAVE_PRINTF PRINTF=screen_printf
```

Usually, PRINTF is already defined to void with this line `DEFINES += PRINTF(...)=...`. Check if PRINTF is already defined somewhere else in your Makefile, and comment out this definition so it doesn’t override the one that we just set.

**Warning:** The PRINTF macro is a debugging feature, and as such it is not intended for use in production. When compiling an application for release purpose, please verify that PRINTF is disabled in your Makefile. In other words, in case of release compilation, put back the line `DEFINES += PRINTF(...)=...` and comment out the other one.

**Warning:** The PRINTF macro can only be used in between successive calls to `io_exchange`. Calling it outside of it will result in unexpected behavior. Behind the scene, PRINTF sends a status on the SEPH. Only one status can be sent in a row, otherwise the SEPH crashes. For this reason, don’t use PRINTF just after status-sending calls, such as UX_DISPLAY. This macro packs a call to `io_seproxyhal_display` and is often the reason for application crashes. Then is no other work around than to move your call to PRINTF somewhere else in your code.

This macro can be used in your code like a classical printf function from stdio.h. However, it is improved with a neat feature to easily print byte arrays:

```c
uint8_t buffer[4] = {0xDE, 0xAD, 0xBE, 0xEF};
```
// PRINTF(string, array length, array);
// .H for uppercase, .h for lowercase
PRINTF("What a lovely buffer:\n %.H \n\n", 4, buffer);
PRINTF("I prefer it lower-cased:\n %.h \n\n", 4, buffer);

user@home:~/work/usbtool$ ./usbtool -v 0x2c97 log
What a lovely buffer:
 DEADBEF
I prefer it lower-cased:
 deadbeef

Fig. 2: Result of the example code printed inside a terminal

20.2 Console Printing

The PRINTF macro triggers messages from the MCU to the host computer through the USB link. We use USBTool to read these messages and print their payload in a terminal.

Unzip the file and execute this command: ./usbtool -v 0x2c97 log

Now you can launch your app on your Nano S, and every PRINTF will end up printed on the host computer, allowing you to debug your program more easily.

20.3 PIN bypass

In Ledger app development, it is necessary to enter your PIN code each time you install an unsigned app. In order to do testing during development, this means developers wind up using many painful button presses entering a PIN code compared to relatively few testing their own application code. The Ledger OS (BOLOS) supports installing a custom developer certificate. By installing a custom certificate once on your device you can avoid having to retype your PIN each time you adjust your app. Here are the steps for the Ledger Nano S:

1. Generate a public / private keypair using the following command:

   ```
   foo@bar:~$ python -m ledgerblue.genCAPair
   Public key :
   0495331cb86e961fc9cb5792a97737e4204b58be99321dcec463cec3057b3304e9875614004e6e540ab0610a1339fa5f71e4
   Private key: 6c1f1df852255e113b23c2e977d6b5c3ea2aaf411f05a5fdab87a3e8f44468ee
   ```

2. Enter recovery mode on your Ledger Nano S. Do this by unplugging it then holding down the right button (near the hinge, away from USB port) while plugging it in again. Recovery mode should then appear on the screen. Enter your pin and continue.

3. Load your public key onto the Ledger Nano S. Paste the public key generated at step 1 after --public. You may need to adjust the --targetId constant to match your device. Find the right targetId for your device here. Notice that we must include a --name parameter containing the name of the custom certificate (any string will do):

   ```
   python -m ledgerblue.setupCustomCA --targetId 0x31100004 --public yourPublicKey --name dev
   ```
If you receive the error `Invalid status 6985` then please review this link and then go back to step 2. The above command is the simplest that can work however some developers may wish to use the optional `--rootPrivateKey` option to specify a secure channel encryption key (which is the private key generated at step 1) or use the `--name` option for convenient labeling according to local convention.

4. Once you have loaded your custom certificate, you can try to load an app you compiled yourself onto your Ledger to see if you are able to bypass the PIN. Before you try it, set the `SCP_PRIVKEY` environment variable to contain the private key generated at step 1 in your shell or `.bashrc`:

```
export SCP_PRIVKEY=yourPrivateKey
```

and then rebuild/load your app.

For more information see `loadApp.py`

**Warning:** A side effect of installing a custom CA on your device is that it will from now on fail to pass the Ledger Genuine Check, which is required to install applications from the Ledger Live. To make it genuine again, you should uninstall your custom CA and all the applications installed through it.

Uninstalling a custom CA is very quick:

1. Enter recovery mode on your Ledger Nano S. Do this by unplugging it then holding down the right button (near the hinge, away from USB port) while plugging it in again. recovery mode should then appear on the screen. Enter your pin and continue.

2. Type this command in your terminal:

```
foo@bar:~$ python -m ledgerblue.resetCustomCA --targetId 0x31100004
```

Find the right `targetId` for your device [here](#).
The Ledger Manager is the service which centralizes the distribution of BOLOS applications on Ledger devices. It is the place where both Ledger’s apps and third party apps are distributed to users. This document aims at explaining the procedure for a third party app to be released on the Ledger Manager.

First, let’s see the typical end-to-end architecture of a BOLOS application, in the case of an app for a cryptocurrency:
In this example, 3 pieces of software are required:

1. A BOLOS application, written in C, running on a Ledger device
2. The companion app, with business logic and typically a GUI, running on a computer or a phone connected to the Ledger device
3. A transaction explorer that interfaces between the front-end and the cryptocurrency daemon, running either locally (in the case of a full node) or remotely (light wallet)

If you’re not developing an app adding support for a cryptocurrency, then you can ignore the 3rd piece. Depending on your use case, the companion app might be a simple daemon or a command line tool, without a GUI. *The companion app should always be compatible with Windows, MacOS and Linux*

When these pieces are ready for deployment, they must be sent to Ledger for a security review. One of the things we’re going to do is review the security of the BOLOS application code, and check that it functions as intended in conjunction with its companion app. Here is a non-exhaustive list of what we’re looking at during the review process:

- Backdoors
- Buffer overflows
- Correct use of the device (for example, minimizing NVRAM wearing)
- Correct use of the SDK
- Application stability, correct handling of errors
- Correct usability (see Design Guidelines)

Review time depends on Ledger’s available resources and code complexity. The more unique and complex the code, the longer the review. Please note that forks of supported BOLOS apps will require a new review. The decision to reject or revoke an app is at Ledger’s sole discretion. We will make our best efforts to provide a rationale for such decisions. Upon fixing identified issues, app authors will be allowed to re-submit their apps. A new all-in-one native app called *Ledger Live* has been released on the 9th of July, 2018. This new platform will be open to pull requests for adding support for new cryptocurrencies in the coming months, *it is not available yet*. Currently, your best shot is to build your own companion app with its own backend. It is less convenient for users than a full integration to Ledger Live, but on the other hand, it gives you more freedom on the design of the interface you want users to see, and the way it will be implemented (Web app? Native app? It’s up to you!).
21.1 Design Guidelines

The design of your device app will be added to our website to inform that your app is available. In that respect, we thank you to provide us with the design in a format and with measurements that shall meet the criteria communicated by our team.

There are 2 mandatory icons that should be provided for Ledger to release an app:
- The Device icon, displayed on the Nano S dashboard
- The Manager icon, displayed in the Manager application list

Below are the Adobe Illustrator templates for such icons, please respect their guidance.
- The color palette for Blue icons shall contain 16 colors maximum
- The icon file shall not contain any alpha channel
- In order to blend well with the Blue’s Dashboard background, the 4 rounded white corners of the blue icon shall be set to this RGB value: #F9F9F9 (and not #FFFFFF)
- Nano X icons must be 14x14 px in size. Developers can create their Nano X icons from the 16x16 px Nano S one with the following command: `convert nanos_app_<token>.gif -crop 14x14+1+1 +repage -negate nanox_app_<token>.gif`

Click on each image to download the corresponding .ai file, and fill them directly before sending them back. They should correspond to the icons used to compile your app.

Most of the time, Bolos apps expose few functionalities to the user. However, despite this simplicity, it is still challenging to build a user-friendly interface, because of the few inputs (only 2 buttons!) and also because of the screen
Fig. 4: Device icon template (click to access Illustrator file)
Fig. 5: Manager icon template (click to access Illustrator file)
size which offer only 128*32 pixels. Ledger provides some guidance to avoid third party developers to reinvent the wheel.

**Note:** See the *BAGL Toolkit* for more information on the BOLOS Application Graphics Library technical details.

Even though the screen is small, you don’t want to bloat it with plenty of tiny text to fit everything in a single time. Instead, multiple successives screens are the way to go, with a configurable time interval between each screen switch. Doing this allows you to split and segment the information to display. It is also a good practice to inform the user of what is the kind of information currently displayed, by giving it a title. In some cases, even when the information is segmented by type, there won’t be enough space for it to fit entirely on screen. Rather than splitting this information in multiple screens, it is better to use automatic scrolling display. Here is an example that sums it up:

![Fig. 6: Example: transaction confirmation screen](image)

On this example two pieces of information are shown: an amount of bitcoins, and a destination address. The two screens are alternating periodically until the user validates or cancel. A title is present to describe each information. For the too-long-to-display destination address, the automatic scrolling is used.

A few graphical pointers (glyphs) help users to make a link between the right and left buttons and their effects. There are four principal functions fulfilled by pressing buttons:

- Making a binary choice for a question asked by the device (1 button)
- Moving in a menu (1 button)
- Selecting a menu entry (2 buttons)
- Proceeding with a disclaimer (2 buttons)

To propose a binary choice to the user, 2 glyphs representing the 2 choices should be placed on each side of the screen (a cross symbol to cancel and a validation symbol to accept). You can display a question and relevant information in between the glyphs, like in the upper example.

To move inside a menu, be it vertical or horizontal, the same technique is applied with different glyphs. We recommend using arrows: up and down for a vertical menu, left and right for a horizontal one).

![Fig. 7: Example use of arrows in an horizontal menu](image)

Once positioned on the right menu entry, the user can select the entry by pressing both buttons. This potential action is not represented by glyphs. Same goes when going through a set of disclaimer screens: only text is displayed, and the user goes to the next part by pressing both buttons. This is a consistent behavior across the device, so you can assume that users are used to it. It should feel obvious when the user has to press two buttons to go through something in your app.
Ledger does not impose the icons nor the interface for third party applications, however user friendliness and guidelines compliance is something being reviewed during the integration process. Therefore, a very poor design might be a cause of delay in your release roadmap if the outcome of the review is negative because of it. Don’t neglect it!

21.2 Design Warranty

The design of your device app shall free from any encumbrances and shall not infringe upon any third party intellectual property right, in particular trademark and design rights. You grant Ledger the right to use such design for free with the right to reproduce and exploit the design for the duration of its display on Ledger website.

You represent and warrant that the device app you are making available on Ledger Live is in compliance with all relevant laws and regulations.

You agree to hold Ledger harmless of any claim arising out of the use of the design and or distribution of the device app.

21.3 Release Process

There are different release levels for a given application:

- **Public release** (official Ledger support)
- **Developer mode release** (experimental support)
  - Developer mode with an **audited** application
  - Developer mode with an **unaudited** application

**Warning:** In any case, Ledger’s decision to publish a third party application may not be considered as any form of partnership nor endorsement of such.

As long as an application isn’t audited by Ledger security team, the message Pending Ledger Review will be made visible when the application is started. The user can acknowledge and skip this warning by pressing both buttons and use the application at his own risks. As an un audited application is considered experimental, such application will be visible only once the developer mode settings has been enabled in Ledger Live settings.

Some applications are also visible once the developer mode setting is enabled, not because of a lack of security review, but rather because they are considered too complex to use. A reason might be a the mandatory compatible GUI is missing, or the application purpose is considered too technical for the vast majority of users.

An application will be listed publicly (official Ledger support) once:
• It passed the security audit
• It’s been tested and validated by at least someone from Ledger Customer Success Team
• The application is compatible with a GUI companion app (CLI is not enough)
• Ledger is ready to communicate officially about the app release (tweet, blogpost, website page update)
• Customer support pages are available on Ledger Help Center

Note: Ledger releases new cryptocurrency apps for its devices whenever reviews, tests, and resources allow it. Applications are usually released on Tuesday or Wednesday.

21.4 Third Party Applications Support

Note: The following only applies if you are adding support for a cryptocurrency and chose to build your own companion app.

Ledger does not provide support for third party applications. **It is required for third party teams to provide and host a complete tutorial that will guide users before the app is released on the Ledger Manager** (please follow these guidelines).

Along with your publication request, app sources and tutorial, please provide information that will allow the Ledger support team to redirect users to relevant support resources:

- **Identity:** Name, Surname, Legal Entity
- **Point of Contact:** URL, email address, phone number

Note: Name, surname and phone number are kept private and will be used only in case of emergency.

21.5 Contact

External developer teams are welcome on Ledger’s Developer Slack. This is the place to get technical support, to discuss Ledger’s tech stack, and more broadly to get in touch with us.

21.6 Warranty and liability disclaimer

The review and publication of third-party Ledger device applications (collectively “the Service”) are provided by Ledger on an “as-is” and “as-available” basis. The Service is subject to change without notice. Ledger disclaims all warranties of accuracy, non-infringement, merchantability and fitness for a particular purpose. To the extent that Ledger makes any pre-release of Ledger device applications available to third-party developers, you understand that Ledger is under no obligation to provide updates, enhancements, or corrections, or to notify you of any changes that Ledger may make, or to publicly announce or introduce the Ledger device applications at any time in the future.

To the extent not prohibited by applicable law, in no event will Ledger be liable for personal injury, or any incidental, special, indirect, consequential or punitive damages whatsoever, including, without limitation, damages resulting from delay of delivery, for loss of profits, data, business or goodwill, for business interruption, or any other commercial damages or losses, arising out of or related to this agreement or your use or inability to use the service.
21.7 Deliverables summary

Please apply on Ledger’s Listing Program:

- Bolos app Release Candidate source code (preferably a git repository)
- Companion app (binaries or package, for Windows/MacOS/Linux)
- Adobe Illustrator templates filled with your icons (see Design Guidelines)
- Contact information (Name, Surname, Legal Entity, URL, email address, phone number)
- Link to tutorial hosted on third party website (see Third Party Applications Support)
- Video of your application running on the Ledger device
  - Verify public address on the Ledger device
  - Display transaction information before allowing signature
  - Reject a transaction on the Ledger device
  - Sign a transaction on the Ledger device

Note: Ledger will review new applications and Pull Requests on a best-effort basis. Submitting an application or a Pull Request isn’t a guarantee it will be released.
External Documentation

In addition to the documentation provided in this hub, we also have the following resources available:

- **BOLOS Python Loader**
Developing applications for Ledger devices (Nano S, X, Blue) is an intricate process. The security of the user funds relies on the fact that the application works in a correct and secure manner and that potential attackers cannot misuse it to extract private data and/or sign requests which are not authorized by the user. The app should guard against such attacks because they have a very low entry point – a Ledger device attached to a compromised host might be a victim of the attacker’s program sending invalid/non-standard requests to the device.

This guide is meant to be a summary of all important aspects of Ledger Apps security and it shall be read by developers before developing an app for Ledger. The guide is divided into multiple sections, each taking on a different aspect of security.

23.1 Development practices

Whenever writing a secure Ledger app, the following advice should not be taken lightly.

23.1.1 Code reviews

We encourage all written code to be peer-reviewed. Importantly, the review should have at least:

1. One reviewer proficient in C and knowing C security weaknesses.
2. One reviewer with “hacker’s mind” (looking at the code from the perspective of an attacker).

23.1.2 Security audits

We encourage third-party security reviews. Note, however, that solid review takes time and a short review might yield a false sense of security (especially if the reviewing party does not have an extensive knowledge of Ledger code specifics).
23.1.3 Developing First App

Apart from reading developer documentation at https://ledger.readthedocs.io/en/latest/ we recommend looking at Sia app https://github.com/LedgerHQ/ledger-app-sia which provides a nice starting point for an app, including lots of explanatory comments. A sample of security-wise overly-paranoid app is https://github.com/LedgerHQ/ledger-app-cardano.

23.2 Cryptography

This section presents general concepts about cryptography development, but also guidelines specific to the security model of the Ledger devices. It gives guidelines to:

- Ensure a potential vulnerability in one application will not cause damages to other apps.
- Make sure all the operations that manipulate secrets are approved by the user.
- Restrict the use of these secrets by apps.

23.2.1 Own crypto primitives

You should never roll your own crypto primitives (including encryption/derivation schemes, hashing functions, HMAC, etc.)

Rationale: It is a purpose of BOLOS operating system to perform these in a secure manner. Importantly, writing your own crypto primitives is likely to open you to side-channel attacks or other problems. If your primitive is not supported by BOLOS (e.g., some very new cryptography), consult with Ledger developers the possibility of including it in the OS.

23.2.2 Avoid blindly signing data

You should never allow signing of any attacker-controlled message unless it has been verified for structural validity. Importantly, you should never sign a message that might be a hash of transaction.

Rationale: If you allow an attacker to blindly sign a message, she can easily supply a hash of a valid transaction. Your signature could then be used to send an unauthorized transaction.

If you want to sign user-supplied “personal” messages, prefix them with a fixed string (which shouldn’t be a valid transaction prefix). It is also a good practice to include message length in the text to be signed. Ledger-app-eth has a good example in function handleSignPersonalMessage. Note that sometimes cryptocurrencies have a standardized way of signing such personal messages and in that case you should use the approved scheme.

Warning: If you allow signing untrusted hashes (while displaying a prompt to the user), be aware that

1. Users do not understand security and could be easily tricked. In fact, they will probably click through your prompt without thinking unless you give them explicit “Warning: this is a very unusual operation. Do not continue unless you know what you are doing” warning. They might not listen even then
2. A compromised host might both change hash on the screen and also data sent to device. This opens the possibility of users signing something they didn’t want to.

23.2.3 Restrict Apps to Coin-Specific BIP32 Prefix

BIP32 paths on which the app can derive keys must be restricted in your application. The chosen derivation paths must not conflict with existing paths if the use case differs.
Warning: Ledger will not sign apps whose BIP32 prefixes have not been properly set.

Restricting the derivation path can be done by setting the `--path` property in the app Makefile.

For example, if your application derive keys on the hardened path 44'/60’, specify in your Makefile:

```
APP_LOAD_PARAMS += --path "44'/60'"
```

Derivation can also be restricted to a specific curve using the `--curve` property. Supported curves are `secp256k1`, `prime256r1` and `ed25519`.

Several curves and paths can be configured. For example, if your app must derive keys on paths 44'/535348’, 13’ and 17’, on curves Ed25519 and prime256r1, the Makefile should contain:

```
APP_LOAD_PARAMS=--curve ed25519 --curve prime256r1 --path "44'/535348'" --path "13’" --path "17’"
```

Rationale: Setting prefixes is crucial, as it limits the amount of damages an attacker can do if he manages to compromise an application. If a vulnerability is exploited on a poorly written of backdoored application, an attacker should not be able to exploit it to extract private keys from other apps, such as Bitcoin or Ethereum keys.

Warning: If your application derives keys on the hardened path 44'/60’ then the chainID parameter must be different from 0 or 1. This is necessary to avoid replaying transactions broadcast on Ethereum-like chains on Ethereum. As a general recommendation, and to ensure a good level of privacy for the end user, we recommend to always use the correct coin type in the derivation path as defined in slip44 (https://github.com/satoshilabs/slips/blob/master/slip-0044.md)

### 23.2.4 Signing/disclosed keys without user approval

Warning: You must always require user approval for signing transactions/messages.

Rationale: If you do not require user consent for signing important data, an attacker can use your device as a signing black box and sign whatever it wants.

**Note:** You might also consider approvals for extracting public keys, as some users might want extended privacy.

1. They might not want to reveal their `root/account` public key, only address keys
2. They might not want to reveal address public key until it is required. (Some cryptocurrencies use addresses that are hash of public keys. It is therefore enough to send the address to the host).

Note that there is a trade-off between privacy and usability here. If you want privacy, it would require a user interaction every time they want to use Ledger device, as opposed to only interaction while signing transactions. The behaviour could also be manually set in the application options.

### 23.2.5 Private Key Management

You should minimize the code that works with private (ECDSA, RSA, etc.) or secret (HMAC, AES, etc.) keys. Importantly, you should always **clear the memory** after you use these keys. That includes key data and key objects.
Leaving parts of private or secret keys lying around in memory is not a security issue on its own because there is no easy way to extract the content of RAM on the chip. If a key is left in RAM by an app, another app will not be able to access it.

However, if the key has not been properly erased, a security issue could lead to the leak of this key, even if it is not used anymore. An attacker able to read arbitrary memory from the app, or execute arbitrary code, will be able to read the content of the stack segment, hence the parts of the key which have not been erased.

A common (and wrong) way of doing this:

```c
uint8_t privateKeyData[64];
cx_ecfp_private_key_t privateKey;

os_perso_derive_node_bip32(
    tmpCtx.transactionContext.curve, tmpCtx.transactionContext.bip32Path,
    tmpCtx.transactionContext.pathLength, privateKeyData,
    NULL);
cx_ecfp_init_private_key(tmpCtx.transactionContext.curve, privateKeyData,
                       32, &privateKey);
explicit_bzero(privateKeyData, sizeof(privateKeyData));

// (later, after privateKey is not needed)
explicit_bzero(&privateKey, sizeof(privateKey));
```

In the happy path, the previous code will correctly clean the memory once the private key is initialized. Note, however, that this code **fails to protect private key in case some system call throws (for example cx_ecfp_init_private_key)**. Correct code should wrap the clearing in `TRY { ... } FINALLY { explicit_bzero() }`.

### 23.2.6 Be Wary of Untrusted Input

Some cryptocurrencies do not have *explicit* fee encoded in the transaction. In such cases, the app cannot rely on fee value sent from the host. Instead, it should receive previous UTxOs and check their output amounts. Note that this usually needs to be done in a separate step due to memory constraints. Check with BTC/Cardano app design for this.

### 23.2.7 Properly protect data you wish to cache on the host computer

Sometimes your app needs to compute over more data than it can fit inside memory. Taking an example from the previous section, it might not be easy to store all UTxOs in memory of Ledger. As such, you might break computation into multiple steps and, for example, verify each UTxO separately and let the host computer to cache a “certified summary”. If you do this, be aware that

1. If the information you want the host to cache is public, you still need to attach a signature to it so that the host cannot send some other value later. This could be done with standard HMAC digest. We would recommend using a temporary (per session) key for this – having a per-session HMAC allows you to truncate the digest size (e.g., you don’t need to have HMAC which withstands years of brute-force attack. Instead, you can balance the digest size against some reasonable upper bound on how long the session lives (e.g., one month should be enough)).

2. If the information is not public, you need to both encrypt and provide a signature. Notably, it is not enough to “scramble” the data by XORing with a random key as this would still allow the attacker to tamper with the values. (Or even break the scrambling if you re-use the same key).

   Instead:
   - Encrypt the information with a sufficiently strong cipher
   - Provide a digest to avoid tampering with the value
23.2.8 Do not allow the host to freely manipulate with key-pairs

Some cryptocurrencies (notably Monero) need to perform an extensive calculation with *(public, private)* key-pair spread over multiple APDU exchanges. If you need to do the same, do not allow the attacker to step out of the protocol. Notably, allowing the attacker to freely perform key manipulation (e.g., group multiplications, exponentiations, etc.) could undermine your app security even if the private key never leaves the device. In general, keep an explicit protocol state machine during the computation. Also, consult with cryptography experts to check for implications if you misstep from the protocol.

23.3 C is your enemy

23.3.1 Know your C compiler

Ledger apps are written in C. Unlike typical embedded project, the goal here is to write for a single platform with a single compiler.

The current supported compiler is clang, and it supports newest language features (up to C11). This is useful for both development and security. You should really learn about the new features and use them extensively as they might help you writing more secure code.

A random collection of useful features: intermingled declarations and code, support of `_Generic`, `_Static_assert`, `_builtin_types_compatible_p`, `__typeof` (very useful for safer versions of macros), etc.

23.3.2 Use safe macro constructs

C is a minefield of problems related to pointers. You can alleviate some of the problems with good macros. However, beware of when they can fail. For example, take the following code:

```c
#define ARRAY_LEN(a) sizeof(a) / sizeof(a[0])
```

Apart from the “obvious” problem of ARRAY_LEN macro being written without external parenthesis, making code such as

```c
(uint8_t) ARRAY_LEN(x)
```

compile to what is understood by the compiler as

```c
(uint8_t) sizeof(x) / sizeof(x[0])
```

there is an important problem with this macro. If used in the function like this

```c
void fn(int x[10]) {
    int len = ARRAY_LEN(x)
}
```

The result is unexpectedly len=1. The reason is that `int x[10]` in the function header is silently converted into `int* x` and the length is therefore `sizeof(ptr) / sizeof(int)` which is indeed 1 on the platform. You can read more about the problem and how to define a safe version in [http://zubplot.blogspot.com/2015/01/gcc-is-wonderful-better-arraysize-macro.html](http://zubplot.blogspot.com/2015/01/gcc-is-wonderful-better-arraysize-macro.html) which explains a patch to the Linux kernel to improve safety of its `ARRAY_SIZE` macro.

Note that similar problems exist, if x is a pointer, with:
memset(x, 0, sizeof(x))

In general, if writing macros, try to write them in a way that they will fail if they get a pointer instead of struct/array.

### 23.3.3 Buffer overflows/underflows

Buffer overflows and underflows are perhaps the biggest source of security vulnerabilities in C code. The following example shows a buffer overflow in (a past version) of one Ledger app.

```c
#define MAX_RAW_TX 200
...
struct tmpCtx {
...
    uint8_t rawTx[MAX_RAW_TX];
} transactionContext_t;

const uint8_t PREFIX[] = {0x00, 0x01, 0x02 ... } // coin-specific signing prefix

void handleSign(uint8_t p1, uint8_t p2, uint8_t *workBuffer,
    uint16_t dataLength, volatile unsigned int *flags,
    volatile unsigned int *tx) {
    ... // (no dataLength validation, we can get up to 255 from APDU)
    if (parseTx(workBuffer, dataLength, &txContent) != USTREAM_FINISHED) {
        THROW(0x6A80);
    }
    ...
    memmove(tmpCtx.rawTx, PREFIX, sizeof(PREFIX));
    ...
    // Here is the vulnerability. There should be a check of
    // if (!(dataLength + sizeof(SIGN_PREFIX) < MAX_RAW_TX)) THROW(...)
    memmove(tmpCtx.transactionContext.rawTx + sizeof(SIGN_PREFIX), workBuffer,
        dataLength);
}
```

In general, there is only a single remedy for buffer overflows - **always check for available buffer space before writing to memory**. The best is to not rely on some specific buffer size but instead rely on sizeof operator. If using sizeof, however, be very careful - if you ever pass a buffer to a function, you are losing its size!

```c
uint8_t a[100];

main() {
    sizeof(a); // 100
    f(a);
    g(a);
}

void f(uint8_t* x) {
    sizeof(x); // 4
}

void g(uint8_t oops[100]) {
    sizeof(oops); // 4
}
```

For the extra safety, consider using a SIZEOF macro defined similarly to [https://github.com/LedgerHQ/](https://github.com/LedgerHQ/).
Be also wary of constructs like

```c
memset(displayAddress, 0, sizeof(fullAddress));
memmove((void *)displayAddress, tmpCtx.address, 5);
memmove((void *)((displayAddress + 5), "...", 3));
memmove((void *)((displayAddress + 8), tmpCtx.address + addressLength - 4, 4));
```

There are several problematic aspects of such code. Apart from truncating important values (see next sections), this code makes a lot of assumptions. For example, if addressLength < 4 (maybe some previous function returns addressLength=0 instead of an error) we get buffer underflow and copy up to 4 bytes of other memory and display it to the user. Or a programmer decides to shorten definition of fullAddress below 13 and we would overwrite memory after the buffer. Finally, this code uses explicit indexes (e.g., 5, 8=5+3). A better trick would be to have a safe helper macro:

```c
#define APPEND(ptr, end, from, len)  
  do {  
    if ((ptr + len >= end || len < 0) THROW(); // not enough space  
      memcpy(ptr, from, len);  
      ptr += len;  
  } while (0)
```

```c
char *ptr_begin = displayAddress;  
char *ptr_end = displayAddress + sizeof(displayAddress); // points behind buffer  
APPEND(ptr_begin, ptr_end, tmpCtx.address, 5); // we should also assert addressLength >= 5  
APPEND(ptr_begin, ptr_end, "...", 3); // Note, we still have explicit size here  
APPEND(ptr_begin, ptr_end, tmpCtx.address - 4, 4); // Note: another assert
```

Finally, you can use SPRINTF macro from `sdk/include/os_io_seproxyhal.h` but be aware that the definition is

```c
#define SPRINTF(strbuf, ...) snprintf(strbuf, sizeof(strbuf), __VA_ARGS__)
```

so the above warning about passing pointers instead of arrays applies to it.

### 23.3.4 Integer overflows/underflows

Integer overflows go hand in hand with buffer overflows. In fact, they can cause serious buffer overflows. Consider following code where a numeric underflow causes buffer overflow of 64kB!

```c
void handleSign(uint8_t p1, uint8_t p2, uint8_t *workBuffer,
    uint16_t dataLength, volatile unsigned int *flags,
    volatile unsigned int *tx)  
#else i;

  // here we don't check if dataLength > 0 so we might be reading behind the buffer  
  tmpCtx.transactionContext.pathLength = workBuffer[0];  
  if ((tmpCtx.transactionContext.pathLength < 0x01) ||
      (tmpCtx.transactionContext.pathLength > MAX_BIP32_PATH))  
  {  
    PRINTF("Invalid path\n");  
    THROW(0x6a80);  
  }
```

(continues on next page)
As a general rule, be very careful about variables which might overflow or underflow. If possible, use bigger types that can accommodate the arithmetic operations you need to perform. For buffer sizes, prefer unsigned types – that way, you can easily check both overflow and underflow in one go, i.e.

```c
void f(uint8_t* buf, size_t bufSize) { // size_t is unsigned
  if (bufSize < REASONABLE_SIZE) THROW(); // guards both against underflow and...
  // overflow!
}
```

### 23.3.5 Data Truncation

Speaking of safely formatting data, be wary of truncated values. Importantly, make sure you do not truncate any important data when displaying on the Ledger screen.

**Example 1:** Truncating tx hash from “f6954eb23ecd1d64c782e6d6c32fad2876003ae92986606585a7187470d5e04” to “f695...5e04” might look nice for the users but this effectively reduces the security of hash and an attacker can now easily try to create a hash collision. Instead, prefer scrolling/paging of long such important values.

**Example 2:** Raise errors instead of truncation

```c
int tmp[10]; // max 10 digits, right?

uint32_t amount = 1987654321
SPRINTF(tmp, "%d", amount) // at least we won't get buffer overflow here ...
display(tmp) // but we display an empty screen!

// but it could be worse
// with bad custom formatting function we could get
format_amount(tmp, SIZEOF(tmp), amount) // "198765432" or "987654321"
```

### 23.3.6 Stack overflow

You application has only a limited size (about ~700B) of stack. That is one of the reasons why stack cookies are not supported yet on the platform.
Given the memory constraints, BOLOS OS does not have memory mapping which would protect from stack overflow errors. As a result, it is very easy to consume more stack space and overwrite the end of your data.

Recommendation:
Enable `DEFINES += HAVE_BOLOS_APP_STACK_CANARY` in your Makefile. This will help you detect stack overflows during app development. If overflow is detected, the app will reboot the device. Note that the overflow check happens only on the next I/O. This means that the protection is not instant and an attacker might avoid the canary check: this option is not a security feature, and has been added to analyze the stack usage during testing process.

### 23.3.7 Optimizations

Do not clear sensitive data with for-loops or other techniques. Do not use `memset` or `bzero` to clear sensitive data: it could be optimized and removed by the compiler.

Recommendation: Use `explicit_bzero` which guarantees that the compiler will not remove the erasure. (See [https://www.owasp.org/index.php/Insecure_Compiler_Optimization](https://www.owasp.org/index.php/Insecure_Compiler_Optimization) for an example of how things could go wrong.)

### 23.4 Business logic problems

#### 23.4.1 Swallowing errors & half-updated states

It goes without saying that you should check return value of functions for any errors. Fortunately, BOLOS throws an error if something goes wrong and you might want to do the same instead of relying on error codes.

There is, however, a more indirect problem. Some BOLOS apps silently catch exceptions in the main event loop without erasing app memory. This could lead into a following insidious bug:

```c
uint16_t totalSize;
uint8_t totalBuf[1000];

void signTx(uint8_t p1, uint8_t p2, uint8_t* data, uint16_t dataSize) {
  if (p1 == P1_INIT) {
    totalSize = 0;
  }

  uint8_t* ptr = totalBuf[totalSize];
  totalSize += dataSize;
  if (totalSize > SIZEOF(totalBuf)) THROW(ERR);
  memcpy(ptr, data, dataSize);

  if (p2 == P2_CONTINUE) {
    THROW(0x9000); // early exit with success
  }
  do_something();
}
```

An attacker might do

1. `signTx(INIT, 100 bytes of data) // OK`
2. `signTx(CONTINUE, 100 bytes of data) // OK`
...
10. `signTx(CONTINUE, 100 bytes of data) // OK`
11. SignTx(CONTINUE, 100 bytes of data) // throws

... 

5. SignTx(CONTINUE, 100 bytes of data) // throws
6. SignTx(CONTINUE, 100 bytes of data) // writes to data[-64..36]

The problem here is that the app state is not updated in a “transactional” manner and attacker exploits this.

Recommendations: Try to not affect global state before you throw. Many times you can use a scratch memory to assemble result and only then do `memmove` to write the result. Even better, wipe memory/reboot device on exceptions to destroy any half-updated app states.

### 23.4.2 Too lenient parsing of transactions

It might happen that your transaction parsing is too lenient. Importantly, this might cause problems if the transaction serialization spec is ambiguous and different clients might interpret it differently. For example, if a field might be repeated one parser might take the first value while another one a second. In general, lenient tx serialization spec should not happen (and if so, the cryptocurrency has bigger concerns than Ledger wallet).

However, even if the network nodes are strict with the serialization checking, you should not slack off on your part. Any ambiguity in parsing adds an attacker a leverage point once it finds some other vulnerability.

Recommendation: Be as strict as possible with transaction parsing. Accept only fields which are in normalized form. If possible, avoid repeated fields and accept fields only in a pre-described order.

Note: you can even go further and do not parse transaction on the device at all. Instead, just send the data in a custom format and let both the app and host serialize the transaction on their own with the app revealing (and signing) only the serialized hash. This way you can avoid bugs in parsing code and be sure both the host wallet and the app agree perfectly on the content of the transaction.

### 23.4.3 Protect Against “Instruction Change” Attacks

Ledger applications live on a secure chip which is very limited in terms of its memory and communication channel. This brings in an interesting problem – the application might not be able to perform all its work in a single request. Instead, the work will need to be spread over multiple requests. Whenever this happens, the application needs to be protected against attacker mixing multiple non-related (or even related) requests.

If your application contains at least one instruction which works over multiple APDU exchanges (e.g., having `P1_INIT/P1_CONTINUE` flag in the standard application “terminology”), you have to protect it from interference. Common attack scenarios:

**Example: Two multi-APDU instructions**

Let’s say you have SignTx and SignMessage, both sharing the same global `hash` variable, both instructions working over multiple APDU exchanges. The attacker might now call

1. SignMessage(INIT)
2. SignTx(INIT)
3. SignMessage(CONTINUE) with data (no finish yet)
4. SignTx(CONTINUE) with data (finish)

At this point, the global memory might be in an inconsistent state (for example, the SignTx hash does contain a different hash than it should be). This might lead to an easy attack.
**Example: Single multi-APDU instruction**

Even if you have only a single instruction with multiple APDU exchanges, an attacker might gain some leverage. Let’s say you have roughly

```c
struct pubkey_ctx {
    int[10] bip32_path;
    int bip32_path_len;
}

struct sign_ctx {
    hash_ctx hash;
    // some other data
}

union {
    pubkey_ctx pubkey;
    sign_ctx signTx;
} ctx;
```

To overwrite the hash context with an exact chosen value.

**Example: “Self”-attack on a single multi-APDU instruction**

You don’t even need two instructions to perform a variation of the attack. Suppose your code goes along these lines

```c
void signTx () {
    if (p1 == P1_INIT) {
        initialize_half_of_my_state();
        if (some_bad_input) THROW(error)
        initialize_rest_of_the_state
    } else {
        // do something
    }
}
```

Assuming that you do not reset state on exceptions, this might happen

1. `signTx(INIT, valid data)`
2. `signTx(CONTINUE, valid data)`
3. `signTx(INIT, data which throws)`
4. `signTx(CONTINUE, more data)`

An attacker now managed to reset half of your state (maybe tx length) but not the rest of it (maybe tx hash) which might allow it to attack your code.

Obviously, there are many variations of this basic scheme and an utmost care needs to be taken here. The recommendation here is:

1. Do not allow mixing of instructions
2. Within instruction, keep an explicit state machine of what is allowed to happen next)
3. Clear memory on exceptions
23.4.4 Use explicit state machines

Whenever a host is required to perform certain actions in a specific order, be sure to explicitly track the state and verify that the next step is consistent. Good examples of what might need to be checked:

1. If host claims some number of tx inputs/outputs, make sure you receive exactly that amount, not more and not less

2. If the host needs to send multiple transaction inputs and outputs and you have to process inputs before outputs, make sure the host cannot send additional input after it received an output.

3. Check that once you finished an action (signing), the attacker cannot resume with additional data (which might be empty). This is important, because usually signing “closes” some hash contexts (or destroy some other data) and re-running SignTx(CONTINUE, empty data) might, therefore, yield either crash or produce a signature of some different data. In general, after finishing a request you should wipe the context variable.

4. If you do not reset UI after sending APDU (for example, because you displayed an address and now you are waiting for another APDU containing tx amount), make sure your button handlers fire just once – a user might press the buttons multiple times. A general recommendation would be to always reset UI with APDU response. Additionally, you can guard your app against itself (and against bad SDK) with tracking whether it should be in IO/UI phase and assert on it in APDU/UI handlers.

An (somewhat contrived) example of problematic button handlers

```c
void handle_sign_message(...) {
    ... // validations
    if (!is_last_apdu) {
        cx_hash(CX_CONTINUE, ctx->hash, data);
    } else {
        memcpy(ctx->last_part, data);
        flags *= IO_ASYNCH;
        display_tx_prompt();
    }
}

unsigned int io_seproxyhal_touch_tx_ok() {
    // for some reason we modify ctx state here
    cx_hash(CX_LAST, ctx->last_part);
    sign(ctx->last_part);
    // now do io_exchange

    // Warning: this might throw (host might do something weird with USB)
    // -> user will press the button second time
    // -> we do another round of cx_hash -> efficiently signing
    io_exchange(...);
    // now reset UI
    ui_idle(); // <-- this line resets button callback
}
```

A fully resilient solution would be

```c
void display_tx_prompt() {
    tx_prompt_handled = false;
    ...
}

unsigned int io_seproxyhal_touch_tx_ok() {
    assert(!tx_prompt_handled);
    tx_prompt_handled = true;
}
```

(continues on next page)
unsigned int io_seproxyhal_touch_tx_cancel() {
    assert(!tx_prompt_handled);
    tx_prompt_handled = true;
    ...
}

But such a solution is needed only if tx_ok or tx_cancel modify context/global variables before calling ui_idle(). (As a side note, if your _ok()/_cancel() handler both 1) do not check whether the memory is cleared, but 2) clear the memory inside the handler; make sure that memclear happens after calling ui_idle()).

Note: If not guarded properly, an attacker might try a following line of attack:

1. Send transaction which is not what user wanted
2. User realizes that the transaction is wrong and presses reject
3. Attacker hogs io_exchange (presumably by doing some bad things to USB communication)
4. User tries again a few more times, thinks that the UI is broken and the app hanged. The callbacks are fired again and again but io_exchange still throws
5. At this point in time, a desperate user might click on “confirm” button to unblock the UI. If an attacker can guess this time, she can un-hog USB and receive confirm callback
Index

C
Custom CA, 26
Custom CA Certificate, 26

D
Device, 26
Device Certificate, 26

I
Issuer, 26
Issuer Certificate, 26

O
Owner, 26
Owner Certificate, 26