Bitcoin High Security Storage Protocol

Version 0.1 Alpha

James Hogan, Jacob Lyles

INCOMPLETE:
DO NOT RELEASE

Alpha Release Warning

This is an ALPHA RELEASE. Though complete and believed to be secure, any unforeseen issues could jeopardize the security of funds stored using this protocol.

This version has not been distributed for public feedback. It is also not distributed with any checksums to verify the integrity of its content.

Disclaimers made, the high-level design has been reviewed both by Bitcoin security experts and the general public, and you'll be hard-pressed to find anything of comparable utility or thoroughness.

Introduction

This document defines a step-by-step protocol for storing Bitcoins in a highly secure manner. It is intended for:

- **Personal storage.** This protocol does not address institutional security needs such as internal controls, transparent auditing, and preventing access to funds by a single individual.

- **Large amounts of money ($100,000+)[1].** It thoroughly considers corner cases such as obscure vectors for malware infection, personal estate planning, human error resulting in loss of funds, and so on.

- **Long-term storage.** It considers not only considers the Bitcoin security landscape today, but also a future world where Bitcoin is much more valuable and attracts many more security threats.

- **Infrequently-accessed funds.** Accessing highly secure Bitcoins is cumbersome and introduces security risk through the possibility of human error, so it is best done infrequently.

- **Technically unskilled users.** Although the protocol is long, it is clear and straightforward to follow. No technical expertise is required.

This protocol will take approximately 4 hours to execute, excluding time to procure equipment and physically store the resulting Bitcoin keys.

If you are already familiar with Bitcoin security concepts and are certain that you want high security cold storage, you may prefer to read Trusting This Protocol and then skip to the section Choosing a Multisignature Withdrawal Policy.

[1] Even if your Bitcoin holdings are more modest, it's worth considering using this protocol. If Bitcoin proves successful as a global currency, it will appreciate 10x (or much more) in the coming years. Security will become increasingly important if your holdings appreciate and Bitcoin becomes a more attractive target for thieves.

The "Protocol Overview" section also describes some lower-security, lower-cost approaches to self-managed storage that may be more appropriate for smaller amounts of funds.

Trusting This Protocol

Funds secured using this protocol can only be as secure as its design. Here's what you can trust about this protocol:

- **Expert advisors:** The development of this protocol was guided with input from Bitcoin technology and security experts. See our advisor list.

- **Open source:** The protocol software is open source. The code is very simple and extensively commented to facilitate easy review for flaws or vulnerabilities.

- **Community review:** The protocol has evolved in conjunction with the wider Bitcoin community -- early versions were circulated during development, and community feedback integrated. See our contributor list.

- **Natural selection:** All documentation and code related to this protocol is under a XXXXXX license, enabling others to publish their own revisions. Inferior alternatives will lose popularity over time.

If you are technically minded, you may review the protocol design document for details on the technical design.

Self-Managed Storage vs. Online Storage Services

Let's start by assessing whether this protocol is right for you.

There there is no such thing as perfect security. There are only *degrees* of security, and those degrees come at a *cost* (in time, money, convenience, etc.) So the first question is: How much security are you willing to invest in?

For most people, most of the time, the authors recommend using a high-quality online wallet service such as Coinbase. It's straightforward to set up, and links to your existing bank accounts and credit cards. Coinbase follows good security practices, is regulated by the US government, and has investment backing from major institutions such as the New York Stock Exchange. Your money is probably safe in Coinbase.

However, all online storage services still come with some notable risks which self-managed storage does not have:

1. **Identity spoofing:** Your account on the service could be hacked (including through methods such as identity theft, where someone convinces the service they are you).

2. **Network exposure:** Online services still need to transmit security-critical information over the Internet, which creates an opportunity for that information to be stolen. In contrast, self-managed storage can be done with no network exposure.

3. **Under constant attack:** Online services can be hacked by attackers from anywhere in the world. People know these services store lots of funds, which makes them much larger targets. If there's a flaw in their security, it's more likely to be found and exploited.

4. **Internal theft:** They have to protect against internal theft from a large group of employees & contractors.

5. **Intentional seizure:** They have the ability (whether of their own volition, or under pressure from governments[2]) to seize your funds.

Some online wallet services have insurance to cover losses, although that insurance doesn't protect against all of these scenarios, and often has limits on the amount insured.

---

[2] There is historical precedent for this, even if funds are not suspected of criminal involvement. In 2010, Cyprus unilaterally seized many bank depositors' funds to cope with an economic crisis. In 1933, the US abruptly demanded citizens surrender almost all gold they owned to the government.

Regardless of how one views the political desirability of these particular decisions, there is precedent for governments taking such an action, and one cannot necessarily predict the reasons they might do so in the future. Furthermore, Bitcoin still operates in a political and legal grey zone, which increases these political risks.

These risks are not theoretical. Many online services have lost customers' funds (and not reimbursed them), including Mt. Gox, Bitfinex, and many more.

Recently, some providers are rolling out higher-security online storage options, such as Coinbase's multisig vault. The design of these services significantly reduces (though does not eliminate) the risks described above, and they are still fairly easy to use. They provide a level of security between that of a typical online wallet and the very high security possible with self-managed storage.

Many people do use online solutions like these to store sizeable amounts of money. We recommend self-managed storage for large investments, but ultimately it's a personal decision based on your risk tolerance and costs you're willing to pay (in money and time) for security.

This protocol focuses exclusively on self-managed storage.

This Protocol vs. Hardware Wallets

Many people who choose self-managed storage (as opposed to an online storage service) use "hardware wallets" such as the Trezor, Ledger, and KeepKey to store their bitcoins. While these are great products that provide strong security, this protocol is intended to offer an even higher level of protection than today's hardware wallets can provide.

The primary security consideration is that all hardware wallets today operate via a physical USB link to a regular computer. While they employ extensive safeguards to prevent any sensitive data (such as private keys) from being transmitted over this connection, it's possible that an undiscovered vulnerability could be exploited by malware to steal private keys from the device.[3]

As with online multisig vaults, many people *do* use hardware wallets to store sizeable amounts of money. We personally recommend this full protocol for large investments, but ultimately it's a personal decision based on your risk tolerance and costs you're willing to pay (in money and time) for security.

[3] For details on this and other security considerations, see the "No Hardware Wallets" section of the protocol design document.

Foundational Concepts

Private Key

Your currency is effectively stored in the Bitcoin blockchain -- the global decentralized ledger. You can imagine a locked box with all of your bitcoins sitting inside of it. This box is unlocked with a piece of information known as “private key”. (Some boxes require multiple private keys to unlock; see the section “Multisignature Security” below.)

Unlike a password, a private key is not meant for you to remember. It’s a long string of gibberish.

The private key is what you need to keep secure. If anyone gets it, they can take your money. Unlike traditional financial instruments, there is no recourse. There is no company that is liable, because Bitcoin is a decentralized system not run by any person or entity. And no law enforcement agency is likely to investigate your case.

Offline Key Storage (“Cold Storage”)

You don’t want to store your private key on any computer that’s connected to the Internet (“hot storage”), because that exposes it to more hacking attempts. There are viruses out there that search computers for private keys and steal them (thereby stealing your money).

One way to protect against this is by encrypting your private key, so even if a thief steals it, they can’t read it. This helps, but is not foolproof. For example, a thief might install keylogger malware so that they steal your password too.

Online keys are inherently exposed to hackers. You therefore need to make sure your private key stays offline (“cold storage”) at all times.

Paper Key Storage

Because the private key is a relatively small piece of information, it can be stored on paper as easily as it can be stored on a computer. And when it comes to key storage, paper has various advantages compared to computers: It’s always offline (no chance of accidentally connecting it to the Internet!), it’s easy & cheap to make multiple copies for backups (and different keys for multisignature security -- see below), and it’s not susceptible to mechanical failure.

For these reasons, this protocol uses paper to store all keys.

Multisignature Security

Central to our security protocols is a technique called “multisignature security.” You’ll need a quick primer on this topic to understand the security protocol.

Regular Private Keys are Risky

Remember that anybody with access to your private key can access your funds. And if you lose your private key, you cannot access your money; it is lost forever. There is no mechanism for reversal, and nobody to appeal to.

This makes it difficult to keep funds highly secure. For example, you might store a private key on paper in a safe deposit box at a bank, and feel fairly safe. But even this is not the most robust solution. The box could be destroyed in a disaster, or be robbed (perhaps via identity theft), or intentionally seized.[4]

What is Multisignature Security?

To address these issues, Bitcoin provides a way to secure funds with a *set* of private keys, such that *some* of the keys (but not necessarily *all*) are required to withdraw funds. For example, you might secure your bitcoins with 3 keys but only need any 2 of those keys to withdraw funds. (This example is known as a “2-of-3” withdrawal policy.)

The keys are then stored in different locations, so someone who gets access to *one* key will not automatically have access to the others. Sometimes, a key is entrusted to the custody of another person, known as a “signatory.”

This approach of using multiple keys is known as “multisignature security.” The “signature” part of “multisignature” comes from the process of using a private key to access bitcoins, which is referred to as “signing a transaction.” Multisignature security is analogous to a bank requiring signatures from multiple people (for example, any 2 of a company’s 3 designated officers) to access funds in an account.

[4] You can try to mitigate these risks by storing the key yourself, perhaps in a fireproof home safe (as opposed to a bank). But this introduces new risks. A determined thief (perhaps a professional who brings safe-drilling tools on their burglary jobs, or who somehow got wind of the fact that you have a $100,000 slip of paper sitting in a safe) might break into the safe and steal the wallet. Or a major natural disaster might prevent you from returning home from an extended period, during which time your safe is looted.

How Does Multisignature Security Help?

Multisignature security protects against the following scenarios:

- **Theft:** Even if somebody physically breaks into a safe, any one key is not enough to steal the money.

- **Loss:** If a key is destroyed or simply misplaced, you can recover your money using the remaining keys.

- **Betrayal:** You may want to entrust one or more signatories with keys to facilitate access to your funds when you are dead or incapacitated. With multisignature security, entrusting them with a key will not enable them to steal your funds (unless they steal additional key(s), or collude with another signatory).

Choosing a Multisignature Withdrawal Policy

Below are common options for withdrawal policies. You will need to select one before beginning the protocol.

Option 1: Self-custody of keys

Our default recommendation is a 2-of-4 withdrawal policy where you manage all of your own keys (i.e. you do not entrust any to the custody of friends or family). 2-of-4 means there are four keys, and any two of those keys can be combined to access your money.

The keys will be distributed as follows:

- One in a safe at home
- The remaining three in safe deposit boxes or private vaults at different locations

It's important to think about estate planning -- making arrangements for your designated agents to be able to access your funds when you are dead (e.g. for distribution to your heirs) or incapacitated (e.g. to pay medical bills). This usually requires significant legal arrangements to be made in advance.[5]

[5] The most failsafe way to ensure your agents will have access to your safe deposit box is to check with the bank. Standard estate planning legal documents *should* allow your agent to access the box upon your incapacity, and to get into it upon your death. But banks can be fussy and sometimes prefer their own forms.

If you have a living trust, one option may be to have your trust as the co-owner of your safe deposit box. That generally allows a successor trustee to access the box.

Option 2: Distributed custody of keys

Another option is to distribute some of your keys to individuals who you trust ("signatories"). This can offer some advantages:

- **Availability:** If you live in a rural area, there may not be many vaults or safe deposit boxes that are practical to get to.

- **Ease of setup:** It may be simpler to distribute keys to signatories than to find available vaults, travel to them, and set up accounts.

- **Ease of estate planning:** You don't need to make complicated legal arrangements for your signatories to access your funds. They'll have the keys they need to do so.

However, there are significant drawbacks:

- **Privacy:** Other signatories will have the ability to see your balance.[6]

- **Signatory collusion:** Although possessing one key won't allow a signatory to access your funds, two signatories might collude with each other to steal your money.

- **Signatory reliability:** A signatory may fail to store the key securely, or they may lose it.

- **Signatory safety:** Giving your signatories custody of a valuable key may expose them to the risk of targeted physical theft.

- **Kidnapping risk:** If you anticipate traveling in high-crime areas with kidnapping risk, your funds will be at greater risk because you'll have the ability to access them remotely (by contacting your signatories and asking for their keys).[7]

For distributed custody, we recommend a 2-of-5 withdrawal policy. The extra key (5 keys rather than 4 keys in Option 1) ensures estate planning functions even when one of your signatories has lost their key.[8]

---

[6] Technical details: Every private key needs to be packaged with the multisig redemption script (since losing all redemption scripts is just as bad as losing all keys). Redemption scripts, however, allow one to view funds. An alternate version of this protocol could be created using vanilla multisig locking scripts rather than P2SH transactions, which would eliminate the ability of signatories to view balances.

[7] Financially-motivated kidnapping hinges on your ability to access funds to give to the kidnappers. If you are literally unable to access additional funds (because the keys are stored in remote vaults which you must be physically present to access, as opposed to held by friends or family who you can call), kidnappers will have no incentive to hold you.

[8] If you have estate planning arrangements which you are confident will allow your agents to access the keys in *your* custody when needed, you should be fine with 4 keys instead of 5 (two keys going to trusted signatories rather than three). Make sure your executors and signatories know to get in touch with each other when needed.

We also recommend keeping at least two keys in your own custody (in different locations) so that you retain the ability to access your own funds.

Protocol Overview

This section establishes a basic understanding of the protocol in order to facilitate its execution. For more background on the protocol's design, see the design document.

As described previously, the general approach involves putting bitcoins in cold storage, using multisignature security, with the keys stored only on paper.

Eternally Quarantined Hardware

This bulk of the protocol consists of ways to safeguard against theft of private keys due to malware infection. To accomplish this, the protocol uses *eternally quarantined* hardware.

Quarantined hardware means we drastically limit the ways in which a piece of hardware interfaces with the outside world in order to prevent the transmission of sensitive data (e.g. private keys) or harmful data (e.g. malware). We consider *all* interfaces -- network, USB, printer, and so on -- because *any* of them might be used to transmit malware or private keys.

Eternally quarantined hardware means we use factory-new hardware for this purpose (to minimize risk of prior malware infection), and *never* lift the quarantine. The quarantine is permanent because any malware infection which *does* somehow get through the quarantine might wait indefinitely for an opportunity to use an available interface (e.g. the Internet, if a quarantined laptop is later used to access the web). Eternal quarantining renders the hardware essentially useless for anything else but executing this protocol.

Parallel Hardware Stacks

There is a class of attacks which rely not on *stealing* your sensitive data (e.g. private keys), but in subverting the process of *generating* your sensitive data so it can be more easily guessed by a third party.[9] We call this "flawed data."

Because we are generating our data in eternally quarantined environments, any malware infection attempting this is unlikely to have come from your other computers -- it would likely have already been present when the quarantined system arrived from the manufacturer.[10]

The way to defeat these attacks is to *detect* them before we actually use the flawed data. We can detect such an attack by *replicating* the entire data generation process on *two* sets of eternally

[9] For example, a variant of the Trojan.Bitclip attack which replaces keys displayed on your screen (or keys stored in your clipboard) with insecure keys.

[10] For example, the Lenovo rootkit or this Dell firmware malware infection.

quarantined hardware, from *different* manufacturers. If the process generates identical data on both sets of hardware, we can be highly confident the data is not flawed.[11]

Bitcoin Core and ProtocolScript

The protocol uses the Bitcoin Core software for all cryptographic and financial operations, as its open source code is the most trustworthy. This is due to its track record of securing large amounts of money for many years, and the high degree of code review scrutiny it has received.

The protocol also utilizes ProtocolScript, a software program that automates much of the manual work involved in executing the protocol. ProtocolScript is small, simple, and has been code reviewed by security experts (see Appendix F).

Protocol Output

The end result of this protocol is a set of information packets, one for each private key needed for the multisignature withdrawal policy. Each packet includes the following information, written or printed on physical paper:

- A brief instruction sheet
- The private key -- a long string of letters and numbers
- A “redemption key” -- an additional (very long) code needed to access funds, identical for all private keys

[11] ...because it would have to be an identical attack present on both sets of hardware, factory-new from different manufacturers. This is exceptionally unlikely.

Lower-security Protocol Variants

The protocol requires over $800 in equipment. If you are willing to accept lower security for lower cost, you can do so with only slight modifications to the protocol:

1. The risk of flawed key generation attacks on eternally quarantined hardware is quite low, and the extra hardware is somewhat costly. If you are willing to accept this risk, you could forego buying the second hardware stack (and needing the second setup computer) and skip the process of re-generating and verifying keys & transactions.

2. An even lower-security variant is to use nothing but existing computer hardware you already possess, disabling all network connections during protocol execution, instead of purchasing new quarantined hardware. This fails to protect against some malware attacks[12], but provides additional savings in cost and effort.

These modifications are left as an exercise to the reader. While they would still leave a very respectable degree of security in place, we do not recommend them unless you feel confident you are able to implement them properly.

Out of scope

There's always more one could do to increase security. While this protocol is designed to provide strong protection for almost everyone, some situations (e.g. being the focus of a targeted attack by a sophisticated, well-resourced criminal organization) are beyond its scope.

For some additional security precautions beyond those provided in the standard protocol, see Appendix A.

[12] Such as infection of a laptop's firmware, malware transmitted via USB drive, or certain undiscovered vulnerabilities in the software used by the protocol.

Equipment Required

The protocol has been written and tested around these specific equipment recommendations.

Eternally quarantined hardware: Set 1

- Factory-sealed computer with 2 USB ports[13] [Amazon: 2016 Dell Inspiron 11.6"]
- Two (2) factory-sealed firmware-protected[14] USB drives (2GB+) from the same manufacturer [Amazon: Kanguru FlashTrust 8GB].

Eternally quarantined hardware: Set 2

- Factory-sealed computer with 2 USB ports from a different manufacturer [Amazon: Acer Aspire One Cloudbook 11"]
- Two (2) factory-sealed firmware-protected USB drives (2GB+) from the same manufacturer, but a *different* manufacturer than the drives for Set 1 [Amazon: IronKey Basic D250 2GB]

Other Equipment

- Two (2) used / existing computers with Internet connectivity and about 2GB of free disk space
- Two (2) *unused* regular USB drives (2GB+) [Amazon: Verbatim 2GB]
- Precision screwdrivers [Amazon], for removing WiFi cards from laptops
- Electrical tape [Amazon]
- Casino-grade six-sided dice.[15] [Amazon] (Regular dice are insufficient.[16])

---

[13] We'll be using two USB drives at the same time. If the computer has only one USB port, you'd need to use a USB hub, which is a separate piece of USB hardware subject to malware infection in its firmware.

[14] Firmware-protected drives defend against malware targeting USB device firmware, such as BadUSB or Psychson.

[15] Standard software algorithms that generate random numbers, such as those used to generate Bitcoin private keys, are vulnerable to exploitation, either due to malware or algorithmic weakness (i.e. they often provide numbers that are not truly random). Dice offer something closer to true randomness.

[16] Regular dice can produce highly biased (and therefore less secure) results.

- Faraday bag [Amazon]. Used to prevent smartphone malware from stealing sensitive data using radio frequencies.

- Table fan [Amazon]. White noise can prevent malware on nearby devices from stealing sensitive data using sound.

- Home safe [Amazon]. Consider bolting it to your floor to deter theft.

- TerraSlate paper [Amazon].[17] Waterproof, heat resistant, and tear-resistant.

- Tamper-resistant seals [Amazon]

[17] TerraSlate paper is extremely rugged, but you might also consider laminating the paper for additional protection. You'll need a thermal laminator [Amazon] and laminating pouches [Amazon].

Protocol Preface

Protocol Structure

The overall protocol consists of four distinct subprotocols:

- **Setup:** Prepares hardware, and downloads and verifies needed software & documentation.
- **Deposit:** For securely storing bitcoins.
- **Viewing:** For viewing the balance of your funds in secure storage.
- **Withdrawal:** For transferring some or all of your stored funds to another bitcoin address.

Sensitive data

Sensitive data (e.g. private keys) will be highlighted in red, like this: sensitive-data-here. Sensitive data can be used by thieves to to steal your bitcoins. If you follow the protocol precisely, your sensitive data will remain secure.

Do *not* do anything with sensitive data that the protocol does not specifically instruct you to do. In particular:

- Never send it over email or instant messenger
- Never save it to disk (hard drive, USB drive, etc.)
- Never paste or type it into any non-eternally-quarantined device
- Never take a picture of it
- Never let any untrusted person see it

Terminal Usage

Many protocol steps involve typing commands into a *terminal window*. Working in a terminal window is analogous to working under the hood of a car. It allows you to give the computer more precise commands than you can through the regular interface.

Commands to be entered into a terminal window will be displayed in a fixed-width font like this:

```
$ echo "everything after the $ could be copy-pasted into a terminal window"
```

The `$` at the beginning of the line represents a *terminal prompt*, indicating readiness for user input. The actual prompt varies depending on your operating system and its configuration; it may be `$`, `>`, or something else. Usually the terminal will show additional information (such as a computer name, user ID and/or folder name) preceding every prompt.

In the above example, the text splits across two lines only because of the margins of this document. Each line is *not* a separate command; it is all one command, meant to be entered all at once. This is clear because there is no terminal prompt at the beginning of the second line.

Proceed Carefully

If you encounter **anything that is different** from what the protocol says you should expect, **the recommendation is that you stop and seek help,** unless your expert opinion gives you high confidence that you understand all possible causes and implications of the discrepancy.

In general, follow the protocol carefully, keep track of what step you are on, and double-check your work. Any errors or deviations can undermine your security.

Setup Protocol, Section I:
Verify and Print Protocol Document

The Setup Protocol is used to prepare hardware, and download and verify needed software & documentation.

The first thing we need to do to verify the integrity of the protocol document (the one you are reading) to ensure that it has not been tampered with.

After verifying the document, we'll print a hard copy. Printing has two benefits:

- A verified *electronic* copy will not be accessible at all times during protocol execution due to reboots and other changes to the computing environment. Printing ensures you always have a verified copy available.

- A printed copy can be used as a checklist, reducing the risk of execution error by ensuring you don't lose your place.

If you do not have access to a printer, you can proceed if you are willing to accept the risk.

1. Download the necessary files by right-clicking the links below, and selecting "Save Link As…" or "Download Linked File".
    a. Latest version of the protocol document (link)
        i. If you already have a copy of the protocol document sitting in your downloads folder, rename it to something else first.
    b. Protocol fingerprint file, used to cryptographically verify the document: (link)
    c. Protocol "public key," used to cryptographically verify the fingerprint file: (link)

2. Download and install GnuPG, the software we'll use for doing the cryptographic verification. GnuPG is the same software recommended by the Electronic Frontier Foundation's Surveillance Self Defense protocol.[18]
    a. **Windows:** (link)[19]

[18] Technical details: Note that we are foregoing verification of the integrity of GnuPG itself. Verification requires having access to a pre-existing, trusted installation of GnuPG, and for many protocol users, this will not be easy to come by. If you *do* have access to a trusted installation of GnuPG, and understand how to do the verification process, we encourage you to do so.

The risk of an unverified PGP installation is relatively small, since an attacker would have to compromise not just the hosting of GPG distributions, but also the hosting of other software distributions used by this protocol, and such a breach would be quickly detected by the global community.

[19] You can view the developer's Windows download page here.

b. **macOS:** (link)[20]
c. **Linux:** GnuPG comes pre-installed with Linux distributions.

3. Open a terminal window.
    a. **Windows:** Press Windows-X and select Command Prompt from the pop-up menu.
    b. **macOS:** Click the Searchlight (magnifying glass) icon in the menu bar, and type "terminal". Select the Terminal application from the search results.
    c. **Linux:** Varies; on Ubuntu, press Ctrl-Alt-T.

4. Change the terminal window's active folder to your downloads folder. The commands below are based on common default settings; if your downloads folder is in a different place, you may need to customize this command.
    a. **Windows:** `> cd %HOMEPATH%/Downloads`
    b. **macOS:** `$ cd $HOME/Downloads`
    c. **Linux:** `$ cd $HOME/Downloads`

5. Verify the integrity of the downloaded document.[21]

    a. You'll need the protocol's "public key" to cryptographically verify the software.
        i. First, verify the signing key fingerprint:
        ```
        $ gpg --with-fingerprint jacob_lyles.asc
        ```

        ii. This key fingerprint is `0591829806AD8C4B11 6856562144D7A3841C6F90`. Sometimes this value is printed with spaces in it. That is okay. Double check this value at https://keybase.io/jacoblyles to make sure this document has not been altered to include a different key.

        iii. Next, import the key:
        ```
        $ gpg --import jacob_lyles.asc
        ```

    If you are ever using the protocol in the future and notice that this step has changed (or that this warning has been removed), there is a security risk. Stop and seek assistance.[22]

---

[20] You can view the developer's macOS download page here.
[21] For technical background about this process, see https://en.wikipedia.org/wiki/Digital_signature.

[22] Technical details: There's a chicken-and-egg problem here, in that this document is giving instructions for how to verify itself. Any attacker that compromised this document could also compromise these instructions so that the verification (erroneously) passes. There's no way to prevent this, unless a reader is familiar with the document *before* the compromise and recognizes that the verification instructions have changed.

In the unfortunate event we legitimately need to publish a new public key (or change the means of obtaining it), we'll first disseminate a public announcement, signed at a minimum with our personal keys, and hopefully with the keys of well-known individuals from the Bitcoin community.

b. Use the public key to verify that the fingerprint file is intact:

```
$ gpg --verify SHA256SUMS.sig SHA256SUMS
```

Expected output:

```
gpg: Signature made Wed Jan  4 17:39:37 2017 PST using RSA key
ID 841C6F90
gpg: Good signature from "JACOB LYLES <jacob.lyles@gmail.com>"
[unknown]
gpg: WARNING: This key is not certified with a trusted
signature!
gpg:          There is no indication that the signature belongs
to the owner.
Primary key fingerprint: 0591 8298 06AD 8C4B 1168  5656 2144
D7A3 841C 6F90
```

The warning message is expected, and is not cause for alarm.[23]

c. Verify the fingerprint file matches the fingerprint of the downloaded ProtocolScript:

```
$  sha256sum -c SHA256SUMS 2>&1
```

Expected output:

```
xor.py: OK
```

6. Switch to use the new document.
    a. Open the version of the document that you just verified.
    b. Close this window (of the unverified version of the document you *had* been using).
    c. Delete the old, unverified copy of the document.

7. Print the verified document.

[23] Technical details: GPG was designed on the premise that public keys would be verified as actually belonging to their owners -- either directly, by receiving a key face-to-face from someone known to you, or indirectly, via cryptographic signature by someone whose public key you've already verified. The warning message merely indicates that you have done neither of these verifications for the protocol's public key.

This is standard practice with software distribution, even for major software packages like Ubuntu. Although you do not have the opportunity to personally verify the protocol's public key came from the protocol authors, you can nonetheless have some degree of trust in the validity of the key, to the extent you trust it was generated and is hosted in a secure manner, and that someone in the community may have noticed and raised an alarm if it *were* surreptitiously changed by an attacker.

Setup Protocol, Section II:
Prepare Non-Quarantined Hardware

1. Select two (2) computers which will be used as “Setup Computers” to set up USB drives. Both Setup Computers must have Internet access.

2. Using sticky notes, label the two Setup Computers “SETUP 1” and “SETUP 2”.

3. With a permanent marker, label the regular (non-firmware-protected) USB drives “SETUP 1 BOOT” and “SETUP 2 BOOT”.

4. Run a virus scan on the Setup Computers.[24] If you don’t have virus scanning software installed, here are some options:
   - Windows: Kapersky[25] ($39.99/yr), Avira[26] (Free)
   - macOS: BitDefender[27] ($59.95/yr), Sophos[28] (Free)
   - Linux: Unnecessary[29]

5. If the virus scan comes up with any viruses, take steps to remove them.

6. Once you have a clean virus scan, your Setup Computers are ready.

[24] This is probably unnecessary, as the Setup Computer is only used for creating the USB drives, and our protocol for doing that safely is very robust. Still, can’t hurt.
[25] Top-rated by Top Ten Reviews
[26] Top-rated by TechRadar
[27] Top-rated by AV-TEST and Tom’s Guide
[28] Top-rated by Tom’s Guide
[29] See http://askubuntu.com/a/506577

Setup Protocol, Section III:
Prepare Quarantined Hardware

1. Separate your quarantined hardware into two parallel sets. Each set should contain:
   - One laptop
   - Two firmware-protected USB drives

   Each component should be supplied by *different* manufacturers from the other set. i.e. your two laptops should be from two different manufacturers, and the USB drives in one set should be from a different manufacturer than the USB drives in the other set.

2. In each set, label all hardware with a permanent marker. Write directly on the hardware.

    a. Label the laptops ("Quarantined Computers") "Q1" and "Q2".

    b. Label one USB drive from each set with "Q1 BOOT" or "Q2 BOOT". These USBs will have the operating system you'll boot the computer with.

    c. Label the other USB drive from each set with "Q1 APP" or "Q2 APP". These USBs will have the software applications you'll use.

3. **Labeled hardware should only interface with hardware that shares the same label ("Q1", "Q2", or "SETUP 1", or "SETUP 2").** For example:
    a. **Don't** plug a **"Q1"** USB drive into a **"Q2"** laptop.
    b. **Don't** plug a **"SETUP 2"** USB drive into a **"Q1"** or **"Q2"** laptop.
    c. **Don't** plug an **unlabeled** USB drive into a **"Q1"** or **"Q2"** laptop.

4. Quarantine the network and wireless interfaces for both laptops:

    a. Unbox laptop. Do **not** power it on.

    b. Put a tamper-resistant seal over the Ethernet port, if it has one.

    c. Physically remove the wireless card.

        i. For the recommended Dell laptop, Dell's official instructions for doing so are here. A YouTube video showing an abbreviated procedure is here.

        ii. For the recommended Acer laptop, the process is similar to the Dell. Note there are two cover screws hidden underneath rubber feet on the bottom of the laptop.

d. After removing the wireless card, cover the ends of the internal wi-fi antennae with electrical tape.

e. If the computer has separate cards for WiFi and Bluetooth, be sure to remove both. (Most modern laptops, including the recommended Acer and Dell, have a single wireless card which handles both.)

5. Fully charge both laptops.

Setup Protocol, Section IV:
Create Boot USBs

Because the eternally quarantined computers cannot connect to a network, they cannot download software. We'll be using USB drives to transfer the necessary software to them.

We will prepare three (4) bootable Ubuntu USB drives. ("Bootable" means that the Ubutnu operating system will be booted directly from the USB drive, without using the computer's hard drive in any way.)

The *first two* USB drives ("Setup Boot USBs") are the USB drives you labeled "SETUP 1 BOOT" and "SETUP 2 BOOT" in Section II. They will be prepared using your Setup Computers, which may be running Windows, macOS, or something else.

The *last two* USB drives ("Quarantined Boot USBs") are the USB drives you labeled "Q1 BOOT" and "Q2 BOOT" in Section II. They will be prepared using your Setup Computers *while booted off a Setup Boot USB.*[30]

1. Perform the following steps on your SETUP 1 computer.

2. If you are not already reading this document on the SETUP 1 computer, open the version that is saved there.

3. Download Ubuntu by going to this link:
   http://releases.ubuntu.com/xenial/ubuntu-16.04.1-desktop-amd64.iso

   Wait until the download is complete.

4. Open a terminal window.[31]

5. Verify the integrity of the Ubuntu download.

[30] Technical details: The Non-Quarantined OS USBs serve two purposes:

- First, they are used for creating the Quarantined App USBs in the next section, which greatly simplifies the process of doing so because we know it'll always be done from an Ubuntu environment. (We can't use the Quarantined OS USBs for this -- they're eternally quarantined, so they need to be permanently unplugged from their Setup Computer the moment they are created.)

- Second, it will be harder for any malware infections on a Setup Computer's default OS to undermine a Quarantined USB setup process (the malware would *first* have to propagate itself to the Non-Quarantined OS USB).

[31] Refer to Section I for instructions if you've forgotten how.

a. View the fingerprint of the file:[32]
   i. **Windows:** >
   ii. **macOS:** `$ shasum -a 256 ubuntu-16. 04.1-desktop-amd64.iso`
   iii. **Ubuntu:** `$ sha256sum ubuntu-16.04.1-desktop-amd64.iso`

b. The following fingerprint should be displayed:[33]
   `dc7dee086faabc9553d5ff8ff1b490a7f85c379f49de20c076f11fb6ac7c0f34`

   It's not important to check every single character when visually verifying a fingerprint. It's sufficient to check the **first 8 characters, last 8 characters, and a few somewhere in the middle.**[34]

6. Create the SETUP 1 BOOT USB.

   a. Windows

      i.

   b. macOS[35]

      i. Prepare the Ubuntu download for copying to the USB.
         1. `$ cd $HOME/Downloads` [36]
         2. `$ sudo hdiutil convert ubuntu-16.04.1-desktop-amd64.iso -format UDRW -o ubuntu-16.04.1-desktop-amd64.img`
         3. When prompted, enter the computer's administrator password.

      ii. Determine the macOS "device identifier" for the Boot USB.
         1. `$ diskutil list`
         2. Insert the SETUP 1 BOOT USB in an empty USB slot.
         3. Wait 10 seconds for the operating system to recognize the USB.
         4. Once more: `$ diskutil list`

---

[32] Technical details: Because you verified the checksum & checksum signature for this document in Section I, we are omitting the GPG verification of some other fingerprints in the protocol. For a detailed security analysis, see the design document.

[33] You can verify this is the official Ubuntu fingerprint here, or follow Ubuntu's full verification process using this guide.

[34] The way these fingerprints work mathematically is that changing even a single bit in the file will result in a completely different fingerprint, so any modification is easy to detect. For technical details about our specific spot checking recommendation, see the design document.

[35] Adapted from this official guide.

[36] This command changes your terminal's active folder to the "Downloads" folder inside your home folder. If you've changed your default downloads folder, you'll again need to customize this command.

5. The output of the second command should include an additional section that was not present in the first command's output.
    a. This section will have `(external, physical)` in the header.
    b. The first line of the section's SIZE column should reflect the capacity of the USB drive.
6. Make a note of the device identifier.
    a. The device identifier is the part of the new section header that comes before `(external, physical)` (for example `/dev/disk2`).

iii. Put Ubuntu on the SETUP 1 BOOT USB.

1. Enter the following command, **making sure to use the correct device identifier;** using the wrong one could overwrite your hard drive!

```
$ sudo dd if=ubuntu-16.04.1-desktop-amd64.img.dmg
of=USB-device-identifier-here bs=1m
```

2. Wait a few minutes for the copying process to complete.

iv. Verify the integrity of the SETUP 1 BOOT USB (i.e. no errors or malware infection).

1. On your desktop, drag the USB drive to the Trash to "eject" it.
2. Remove the USB drive from the USB slot and immediately reinsert it.[37]
3. Wait 10 seconds for the operating system to recognize the USB.
4. `$ cd $HOME/Downloads`
5. `$ sudo cmp -n `stat -f '%z' ubuntu-16.04.1-desktop-amd64.img.dmg` ubuntu-16.04.1-desktop-amd64.img.dmg USB-device-identifier-here`[38]
6. Wait a few minutes for the verification process to complete.
7. If all goes well, it should return no data.
8. If there is an issue, you'll see a message like:

```
ubuntu-16.04.1-desktop-amd64.img.dmg /dev/disk2
differ: byte 1, line 1
```

If you see a message like this, STOP -- this may be a security risk. Restart this section from the beginning. If the issue persists, try using a different USB drive or a different Setup Computer.

[37] This is a security precaution. See the corresponding step in the Ubuntu section for a detailed footnote.

[38] See the corresponding step in the Ubuntu section for a detailed footnote.

c. Ubuntu[39]

i. Determine the Ubuntu "disk identifier" for the Boot USB you are creating.

1. `$ lsblk | grep disk`
2. Insert the SETUP 1 BOOT USB in an empty USB slot.
3. Wait ten seconds for the operating system to recognize the USB.
4. Once more: `$ lsblk | grep disk`
5. The output of the second command should include an additional line that was not present in the first command's output.
    a. The fourth column for this line should reflect the capacity of the USB drive.
6. Make a note of the disk identifier.
    a. The disk identifier is the first column of the new line, and should start with the letters `sd` (for example `sda` or `sdb`).

ii. Put Ubuntu on the SETUP BOOT 1 USB.

1. Open the Ubuntu search console by clicking the purple circle/swirl icon in the upper-left corner of the screen.
2. Type "startup disk creator" in the text box that appears
3. Click on the "Startup Disk Creator" icon that appears.
4. The "Source disc image" window should show the .iso file you downloaded. If it does not, click the "Other" button and find it in the folder you downloaded it to.
5. In the "Disk to use" option, you should see two lines. They may vary from system to system, but each line will have a disk identifier in it, highlighted in the example below:

    Generic Flash Disk (/dev/sda)

6. Select the line containing the disk identifier you noted above.
7. Click "Make Startup Disk" and then click "Yes".
8. Wait a few minutes for the copying process to complete.

iii. Verify the integrity of the SETUP 1 BOOT USB (i.e. no errors or malware infection).

1. On your desktop, right-click the corresponding USB drive icon in your dock and select Eject from the pop-up menu.
2. Remove the USB drive from the USB slot and immediately re-insert it.[40]
3. Wait 10 seconds for the operating system to recognize the USB.

[39] Adapted from this official guide.

[40] Technical details: In order to avoid detection, it's conceivable that malware might wait until a USB drive is in the process of being ejected (and all integrity checks presumably completed) before infecting the USB. Ejecting and re-inserting the USB *before* integrity checking is a simple workaround to defend against this.

4. `$ cd $HOME/Downloads` [41]
5. `$ sudo cmp -n `stat -c '%s' ubuntu-16.04.1-desktop-amd64.iso` ubuntu-16.04.1-desktop-amd64.iso /dev/USB-disk-identifier-here` [42]
6. If prompted for a password, enter the computer's root password.
7. Wait a few minutes for the verification process to complete.
8. If all goes well, it should return no data.
9. If there is an issue, you'll see a message like:

   ```
   ubuntu-16.04.1-desktop-amd64.iso /dev/sda differ:
   byte 1, line 1
   ```

   If you see a message like this, STOP -- this may be a security risk. Restart this section from the beginning. If the issue persists, try using a different USB drive or a different Setup Computer.

7. Create the Q1 BOOT USB

   a. Boot the SETUP 1 computer from the SETUP 1 BOOT USB.

      i. Reboot the computer.

      ii. Press your laptop's key sequence to bring up the boot device selection menu. (Some PCs may offer a boot device selection menu; see below.)

         1. **PC:** Varies by manufacturer, but is often **F12** or **Del**. The timing may vary as well; try pressing it when the boot logo appears.
            a. On the recommended Dell laptop, press F12. You should see a horizontal blue bar appear underneath the Dell logo.
            b. The recommended Acer laptop does not have a boot menu. See below for instructions.

         2. **Mac:** When you hear the startup chime, press and hold **Option (⌥)**.

      iii. Select the proper device to boot from.

         1. **PC:** Varies by manufacturer; option will often say "USB" and/or "UEFI".

[41] This command changes your terminal's active folder to the "Downloads" folder inside your home folder. If you've changed your default downloads folder, you'll again need to customize this command.

[42] Technical details: When writing the ISO to the USB, there are padding bits written after the end of the files, so the comparison is not, by default, identical. cmp's -n flag allows us to exclude the padding from the comparison by only looking at the first N bytes, where N is the size of the .iso.

a. On the recommended Dell laptop, select “USB1” under “UEFI OPTIONS”.
b. The recommended Acer laptop does not have a boot menu. See below for instructions.

2. **Mac:** Click the “EFI Boot” option and then click the up arrow underneath it. You do not need to select a network at this time.

iv. Some laptops don’t have a boot device selection menu, and you need to go into the BIOS configuration and change the boot order so that the USB drive is first.

1. On the recommended Acer laptop:
   a. Press F2 while booting to enter BIOS configuration.
   b. Navigate to the Boot menu.
   c. Select USB HDD, and press F6 until it is at the top of the list.
   d. Press F10 to save and automatically reboot from the USB.

v. If the computer boots into its regular OS rather than presenting you with a boot device or BIOS configuration screen, you probably pressed the wrong button, or waited too long.

1. Hold down your laptop’s power button for 10 seconds. (The screen may turn black sooner than that; keep holding it down.)

2. Turn the laptop back on and try again. Spam the appropriate button(s) repeatedly as it boots.

3. If the computer boots *immediately* to where it left off, you probably didn’t hold down the power button long enough.

vi. You’ll see a menu that says “GNU GRUB” at the top of the screen. Select the option “Try Ubuntu without installing” and press Enter.

vii. The computer should boot into the USB’s Ubuntu desktop.

b. Enable WiFi connectivity.

i. Click the cone-shaped WiFi icon near the right side of the menu bar.

ii. If the dropdown says “No network devices available” at the top, you need to enable your networking drivers:[43]

1. Click on “System Settings”. It’s the gear-and-wrench icon along the left side of the screen.

[43] Drivers are software that allows the operating system to interface with a piece of hardware.

2. A System Settings window will appear. Click the “Software & Updates” icon.

3. A Software & Updates window will appear. Click the “Additional Drivers” tab.

4. In the Additional Drivers tab, you’ll see a section for a Wireless Network Adapter. In that section, “Do not use the device” will be selected. Select any other option besides “Do not use the device.”

5. Click “Apply Changes”.

6. Click the cone-shaped WiFi icon near the right side of the menu bar again. There should be a list of WiFi networks this time.

7. Select your WiFi network from the list and enter the password.

c. Repeat steps 1-6 using the SETUP 1 computer to create the Q1 BOOT USB rather than the SETUP 1 BOOT USB.

   i. **The instruction to plug a Quarantined Boot USB into your Setup computer *should* raise a red flag for you, because you should never plug a quarantined USB into anything other than the quarantined computer it is designated for!**

      This setup process is the ONE exception.

   ii. Because you have booted the SETUP 1 computer off the SETUP 1 BOOT USB, you will follow the instructions for Ubuntu, even if your computer normally runs Windows or macOS.

   iii. Immediately after you are finished executing steps 1-6 with the Q1 BOOT USB, remove the Q1 BOOT USB from the SETUP 1 computer.
      1. On your desktop, right-click the corresponding USB drive icon in your dock and select Eject from the pop-up menu.
      2. Remove the USB drive from the USB slot.

   iv. **The Q1 BOOT USB is now eternally quarantined. It should never again be plugged into anything besides the Q1 computer.**

8. Create the SETUP 2 BOOT USB and Q2 BOOT USB

   a. Repeat steps 1-7 using the SETUP 2 computer, SETUP 2 BOOT USB, and Q2 BOOT USB.

Setup Protocol, Section V:
Create App USBs

We will prepare two (2) “Quarantined App USB” drives with the software needed to execute the remainder of the protocol. These are the USB drives you labeled “Q1 APP” and “Q2 APP” in Section III.

1. Boot the SETUP 1 computer off the SETUP 1 BOOT USB if it is not already. (See the instructions in Section III for details.)

2. Insert the Q1 APP USB into the the SETUP 1 computer.

    a. **The instruction to plug a Quarantined App USB into your Setup computer *should* raise a red flag for you, because you should never plug a quarantined USB into anything other than the quarantined computer it is designated for!**

    This setup process is the ONE exception.

3. Press Ctrl-Alt-T to open a terminal window.

4. Install the protocol document and ProtocolScript on the Q1 APP USB.

    a. Download the protocol document and ProtocolScript:
    Click https://github.com/jacoblyles/cold_bitcoin/archive/master.zip

    b. Unpack the downloaded ZIP file into a staging area.
        i. When the download completes, Firefox will ask you if you want to open the ZIP file with Archive Manager. Click Yes.
        ii. Extract the ZIP file to a new folder under your home directory called “Protocol”.
        iii. `$ cd ~/Protocol`

    c. Verify the integrity of the download.

        i. You’ll need the protocol’s “public key” to cryptographically verify the software:
        `$ gpg --import jacob_lyles.asc`

        ii. Use the public key to verify that the fingerprint file is intact:
        `$ gpg --verify SHA256SUMS.sig SHA256SUMS`

        Expected output:
        ```
        gpg: Signature made Thu 05 Jan 2017 01:20:10 AM UTC using
        RSA key ID 841C6F90
        ```

```
gpg: Good signature from "JACOB LYLES
<jacob.lyles@gmail.com>"
gpg: WARNING: This key is not certified with a trusted
signature!
gpg:       There is no indication that the signature
belongs to the owner.
Primary key fingerprint: 0591 8298 06AD 8C4B 1168  5656
2144 D7A3 841C 6F90
```

The warning message is expected, and is not cause for alarm.[44]

iii. Verify the fingerprint file matches the fingerprint of the downloaded files:

```
$  sha256sum -c SHA256SUMS 2>&1 | grep OK
```

Expected output:

```
xor.py: OK
```

d. Copy Protocol files to the Q1 APP USB.

5. Open the protocol document so that it is available for copy-pasting terminal commands.

6. Install the remaining application software on the Q1 APP USB.

   a. Configure our system to enable access to the software we need in Ubuntu's "package repository":[45]

      i. `$ sudo mv /var/cache/app-info/xapian/default /var/cache/app-info/xapian/default_old` [46]
      ii. `$ sudo mv /var/cache/app-info/xapian/default_old /var/cache/app-info/xapian/default`
      iii. `$ sudo apt-add-repository universe`
      iv. `$ sudo apt-add repository ppa:bitcoin/bitcoin`
          1. Press Enter when prompted.
      v. `$ sudo apt-get update`

   b. Download and perform integrity verification[47] of software available from Ubuntu's package repository:

[44] For technical details, see the corresponding footnote in Section I.

[45] A "package repository" is roughly analogous to an "app store" on other platforms, although all of the software is free.

[46] Technical details: On Ubuntu 16.04.01 there is a bug in Ubuntu's package manager that affects systems running off a bootable Ubuntu USB. The commands in steps a and b are a workaround.

[47] Integrity verification is done automatically by Ubuntu's apt-get command.

- **bitcoind:** Bitcoin Core, which we'll use for cryptography & financial operations
- **qrencode:** Used for creating QR codes to move data off quarantined computers
- **zbar-tools:** Used for reading QR codes to import data into quarantined computers

```
$ sudo apt-get install qrencode zbar-tools bitcoind
```

c. Copy that software to the Q1 APP USB.

i. Create a staging folder for the files that will be moved to the USB:
```
$ mkdir ~/staging
```

ii. Copy the software into the staging folder:
```
$ cp /var/cache/apt/archives/*.deb ~/staging
```

iii. Copy the contents of the staging folder to the Q1 APP USB:
```
$
```

7. Verify the USB has the correct files:
```
$ ls -1 usb-drive
```

Your output should look similar (but not necessarily identical[48]) to this:[49]
```
bitcoind_0.13.1-xenial2_amd64.deb
libboost-chrono1.58.0_1.58.0+dfsg-5ubuntu3.1_amd64.deb
libboost-program-options1.58.0_1.58.0+dfsg-5ubuntu3.1_amd64.deb
libboost-thread1.58.0_1.58.0+dfsg-5ubuntu3.1_amd64.deb
libdb4.8++_4.8.30-xenial2_amd64.deb
libevent-core-2.0-5_2.0.21-stable-2_amd64.deb
libevent-pthreads-2.0-5_2.0.21-stable-2_amd64.deb
libqrencode3_3.4.4-1_amd64.deb
libsodium18_1.0.8-5_amd64.deb
libzbar0_0.10+doc-10ubuntu1_amd64.deb
libzmq5_4.1.4-7_amd64.deb
qrencode_3.4.4-1_amd64.deb
xor.py
zbar-tools_0.10+doc-10ubuntu1_amd64.deb
```

8. Eject and physically remove the Q1 APP USB from the SETUP 1 computer.

The Q1 APP USB is now eternally quarantined. It should never again be plugged into anything besides the Q1 computer.

[48] Technical details: Version numbers may change, and it's possible some of the packages may be different as well, since dependencies can change from one version to the next. Ubuntu's repositories don't keep old versions of packages accessible, or we'd just have the protocol refer to a fixed version.

[49] The reason the list has more than the software packages we explicitly downloaded is because the software we requested requires the other pieces of software to work properly, so apt-get automatically downloaded them.

9. Repeat all above steps using the SETUP 2 computer, SETUP 2 BOOT USB, and Q2 APP USB.

10. Find a container in which to store all of your labeled hardware, along with the protocol hardcopy, when you are finished.

Setup Protocol, Section VI:
Prepare Quarantined Workspaces

This section is meant to be done immediately before executing the Deposit or Withdrawal protocols.

If you are executing the Setup Protocol for the first time and do **not** plan on executing the Deposit or Withdrawal protocol now, you can stop here.

1. Block side channels

   Side-channel attacks are a form of electronic threat based on the physical nature of computing hardware (as opposed to algorithms or their software implementations). Side channel attacks are rare, but it's relatively straightforward to defend against most of them.

   a. Visual side channel
      i. Ensure that no humans or cameras (e.g. home security cameras, which can be hacked) have visual line-of-sight to the Quarantined Computers.
      ii. Close doors and window shades.

   b. Acoustic side channel
      i. Choose a room where sound will not travel easily outside.
      ii. Shut down nearby devices with microphones (e.g. other laptops). Your smartphone will need to stay on -- it'll be used during the protocol.
      iii. Plug in and turn on a table fan to generate white noise.

   c. Power side channel
      i. Unplug both Quarantined Computers from the wall.
      ii. Run them **only on battery power** throughout this protocol.
      iii. Make sure they are fully charged first! If you run out of battery, you'll need to start over.

   d. Radio and other side channels[50]
      i. Turn off all other computers and smartphones in the room.
      ii. Put them in the Faraday bag and seal the bag.

2. Put your Q1 BOOT USB into an open slot in your Q1 computer.

3. Boot off the USB drive. If you've forgotten how, refer to the procedure in Section IV.

4. Plug the Q1 APP USB into the Q1 computer.

[50] Including seismic, thermal, and magnetic.

5. Open Q1 APP USB, verify you can see the files.

6. Open a Terminal window by pressing Ctrl-Alt-T.

7. Open the protocol document so it's available for reference and copy-and-paste.

   You won't be able to click any external links in the document, since you don't have a network connection. If you need to look something up on the internet, do so in a distant room. Do not remove devices from the Faraday bag before doing going to the other room.

8. Repeat the above steps using the Q2 computer, Q2 SETUP USB and Q2 APP USB.

Deposit Protocol, Section I: Key Generation

The Deposit Protocol is used to transfer bitcoins into high-security cold storage.

1. If this is **not** your first time working with the protocol, check the Release Notes of the latest version of the protocol to see if there are any new versions of the protocol recommended.

2. Execute Section VI of the Setup Protocol to prepare your quarantined workspace.

3. Create entropy for private keys

   Creating an unguessable private key requires *entropy* -- random data. We'll combine two sources of entropy, as this ensures securely random keys even if *one* source is somehow flawed or compromised to be less-than-perfectly random.

   a. Generate computer entropy

   b. Generate dice entropy

Generating Keys and Addresses

This is based off of information given here. The output of the WIF script is checked against the sample WIF app linked off the bitcoin wiki here.

On quarantined computer. Do this five times:

1. Get random seed from the computer's randomness source with the following command
   a. $ xxd -l 32 -p /dev/random | tr -d \\n
   b. Paste the random seeds into a text file for later use
   c. (high security) Write this down. You will need to transfer these exact seeds to the second quarantine computer to double check the computations.
2. Roll your 64 dice and write them down
3. Next you will use the utility script to generate pairs of private keys and addresses
4. First, start bitcoind at the command line.
   a. $ bitcoind -daemon
5. Now run the following command

a. $ xor.py keygen
b. The script will prompt you to type in a dice roll and random seed and it will return a private key and address pair. Paste these to a text file and transcribe them on paper. CASE MATTERS (upper/lower case). Double check what you have written.

At the end of this process, you should have five private key/address pairs.

Generating multisig deposit address

Next you will generate a multisig cold storage address for depositing funds.

1. Run the following command:
    a. $ xor.py multisig-deposit -m 2 -n 5
    b. “m” and “n” can be chosen for different values of m-of-n multisig storage
2. The script will ask for 5 addresses. Use the addresses that you generated in the previous section.
3. The script will return a multisig address and a redeem script. The multisig address is the cold storage address that you will use to deposit funds. The redeem script will be necessary when you eventually want to withdraw funds from cold storage.
4. Transfer the redeem script off with a QR code:
    a. $ qrencode -o redeem.png <paste redeem script here>
    b. $ eog redeem.png
    c. This should open a window displaying a QR code. Use any QR code reading program on another device to read it
    d. Visually verify that your device has read the data correctly. Especially check the first few and last few numbers to make sure you did not make a mistake in copying and pasting
5. Transfer the multisig address off with a QR code.
    a. $ qrencode -o multisig.png <paste multisig address here>
    b. $ eog multisig.png
    c. This should open a window displaying a QR code. Use any QR code reading program on another device to read it
    d. Visually verify that your device has read the data correctly. Especially check the first few and last few numbers to make sure you did not make a mistake in copying and pasting

When ready, you may send coins to the multisig address.

4.

We’ll be rolling dice to generate random numbers used for creating private keys. Proper dice[51] are a more reliable source of randomness than ordinary random number generator software.

[51] Casino dice, as recommended in the Equipment Required section of this document.

a. Roll 64 six-sided dice[52], shaking the dice thoroughly each roll.

b. Write down the results using **pen and paper** -- *not* electronically**.**

c. If you are rolling several dice at the same time, read the dice left-to-right. **This is important.**[53]

d. We'll refer to the complete, 64-digit result as the random seed.

5. Generate a 24-word English private key using the random seed. (The English private key is a private key encoded into English words to minimize the risk of transcription errors.[54])

   a. Type the following at the command prompt:

   ```
   bx mnemonic-new <random seed>
   ```

   Example:

   ```
   bx mnemonic-new 2513231...56223213
                   |-------- 64 digits ---------|
   ```

   b. If you make a transcription error or two when typing in the random seed, it's all right. It won't have any meaningful effect on the security of your keys.[55]

6. Write down the English private key on an *unused* piece of TerraSlate paper. Nothing else should be written on this piece of paper besides this one English private key.

7. Generate a raw private key[56] using the English private key. (The raw private key is used by Bitcoin software.)

   a. Type the following at the command prompt:

   ```
   bx mnemonic-to-seed <English private key> | bx ec-new
   ```

---

[52] The size of the bitcoin address space is 160-bits, which is covered by 62 dice rolls. We use 64 dice rolls because bx requires that the size of seed input be evenly divisible by 32 bits.

[53] Humans are horrible at generating random data and great at noticing patterns. Without a consistent heuristic like "read the dice left to right", you may subconsciously read them in a non-random order (like tending to record lower numbers first). This can drastically undermine the randomness of the data, and could be exploited to guess your private keys.

[54] It's a lot easier to transcribe something like "correct horse battery staple forget sofa..." without errors than "5Kb8kLf9zgWQnogidDA76MzPL6TsZZY36hWXMssSzNydYXYB9KF".

[55] 62 dice rolls creates 10^48 combinations. The fastest supercomputer in the world can do about 10^17 operations per second. Even with the unrealistic assumptions of an attacker controlling this much computing power, and one key guessing attempt taking only one hardware operation, it would take about 10^23 years to crack a private key, whereas the universe is a mere 10^10 years old. There's more than enough buffer to accommodate a few typos (which are themselves fairly random).

[56] We use HD wallets because Electrum requires them for multisignature transactions.

Example:

```
bx mnemonic-to-seed  correct horse ... super flight  | bx ec-new
                     |------------------ 24 words ------------------|
```

Note that the vertical bar is a pipe symbol (shift-backslash on most keyboards), not an uppercase `I` or lowercase `l`.

8. Generate a raw public key using the raw private key. (Public keys are *not* sensitive information, but *will* be required to access your funds.)

    a. Type the following at the command prompt:

    ```
    bx ec-to-public <raw private key>
    ```

    Example:

    ```
    bx ec-to-public
    66bf41baae424a0b91e7b375e1355faef30f6d0da1454efdcedfcaf6cf85203
    c
    ```

9. Copy and paste this raw public key into the text editor window.
    a. All public keys should be pasted together in the same window.
    b. You do not need to keep track of which public key came from which private key.

10. Repeat this entire section for each private key you need to generate. (Five keys total, if you are using our recommended 2-of-5 signature policy.)

11. Transfer each public key off the computer with a QR code. First generate the QR code using the following command:
    a. $ qrencode “<public key>” -o <filename>
    b. An appropriate filename is something like “public_key_1”

12. Open the QR code image using the graphical file explorer or by using the command “$ eog <filename>” at the command line. Scan the QR code with a QR code reader app on another device. Collect the QR codes to send to the trusted principles.

Label the private key on paper

Verify keys on both computers

Verify redemption script on both computers

Repeat N times

Deposit Protocol, Section VII: Address Generation

In this section, a multisignature address will be created using the keys generated in the section above

1. Execute the following command, where <pubkey X> refers to one of the public keys that you created above. This will return a multi-sig address

2. $ bx script-to-address "2 [<pubkey 1>] [<pubkey 2>] [<pubkey3>] [<pubkey 4>] [<pubkey 5>] 5 checkmultisig"

3. This is the address that you will send your bitcoins to for cold storage. Transfer this address off your machine using a QR code. Enter the following command at the terminal. The value entered after “-o” will be the filename of the qrcode image
    a. $ qrencode “<address>” -o address

4. Open the QR code image. From the command line, you can do this with the command “$ eog <filename>”. Use a QR reader on your phone to scan the code on the screen. Use whatever bitcoin software you currently use to send the coins to the address

5. At this point you are done. Turn off the computer. Remove the USB stick. Save these pieces of hardware for the future when you will

Verify addresses on both computers

Deposit Protocol, Section VIII: Initial Deposit

Indirection transaction when depositing

Lay out options of using coin mixers for deposits / withdrawals. Mention risk (trusted third party) / privacy tradeoff.

Do a test withdrawal!

Deposit Protocol, Section IX: Subsequent Deposits

Explain tradeoffs in terms of creating multiple sets of private keys up front (vs. when doing a partial withdrawal)

Deposit Protocol, Section X:
Key Storage

What are the things we need to print?

- Private keys
- Instructions for withdrawing cash (with instructions on how to access redeem script)
- Instructions for depositing cash (with owner's keys only)
- Redeem script (with owner's keys only)

Number each set of private keys with date (and number if multiple being created on date)

Don't record info on who stores the other keys -- safety issue

Seals on printed keys
Don't transmit keys to keyholders electronically, including by phone. (courier is probably fine)

Seals on USBs, computers
Store USBs, and ideally computer, with in safe deposit box

Annual Periodic check on key custody

Before reading QR codes, go into airplane mode. Translate QR to text & verify content before going back online.

Make note of the protocol version used for deposits

Signatories will know how much you have in savings!!

Viewing Protocol

Withdrawal Protocol, Section I: (section name)

The Withdrawal Protocol is used to transfer bitcoins out of high-security cold storage.

1. If this is **not** your first time working with the protocol, check the Release Notes of the latest version of the protocol to see if there are any new versions of the protocol recommended.

2. Execute Section VI of the Setup Protocol to prepare your quarantined workspace.

Withdrawing funds

Eventually, you will want to withdraw funds from your multisig address. To do so, you will first need to gather some information about the deposit transaction using an online computer

1. You can only withdraw funds that are associated with one particular deposit transaction at a time. On your online laptop, search for your multisig address on https://blockchain.info to see the transactions that have been sent to it. Pick one and copy the txid to someplace where you will have easy access to it
2. You will need some data from this deposit transaction in order to generate the withdrawal transaction. There are a variety of ways to get this data, but the easiest way is to paste your txid into https://chainquery.com/bitcoin-api/getrawtransaction and select the “decoded” option and then submit the form.
3. The deposit transaction might have multiple outputs in the “vout” section. Find the output where the “addresses” array has your multisig address in it. You will need a few pieces of data from this output:
    a. “n” (later referred to as “vout”)
    b. “value”
    c. The “hex” value of “scriptPubKey”

You will complete the following steps on your quarantined computer to generate and sign the withdrawal transaction:

1. Transfer the following data to the quarantined computer. For long or critical data, such as the redeem script and the destination address, step 2 will tell you how to transfer it using QR codes
    a. Deposit transaction data:
        i. txid
        ii. vout
        iii. value
        iv. scriptPubKey hex

    b. m of the private keys
    c. The multisig cold storage address
    d. The redeem script
    e. The destination address
        i. CRITICAL: Make sure you copy the destination address correctly. If there are any errors, your funds will be lost!
2. For long or critical data, such as the redeem script or the destination address, transfer the data to the quarantined computer using a QR code instead of typing it in using the following process
    a. Make a QR code of the desired data using an online tool like http://goqr.me/ or a mobile app
    b. Enter
3. At the command line, run the following command
    a. $ xor.py multisig-withdraw
4. The script will ask you for the data above.
5. [TODO fee guidance]You will be asked to choose a transaction fee. Figure out what the fee for a normal transaction is and use that.
6. Any amount leftover from the withdrawal transaction will be sent back to the source address as change. The formula is:
    a. change = input value - withdrawal value - fee
7. The script results in a long hex string representing the signed transaction. It should also tell you that the transaction is “complete” if it has been signed with enough keys.
8. Move the hex string representing the signed transaction off the computer using a qr code
    a. $ qrencode -o signed_tx.png <paste transaction string here>
    b. $ eog signed_tx.png
    c. This should open a window displaying a QR code. Use any QR code reading program on another device to read it
    d. Visually verify that your device has read the data correctly. Especially check the first few and last few numbers to make sure you did not make a mistake in copying and pasting

Now that you have the signed withdrawal transaction, it will have to be broadcast to the bitcoin network to be included in the blockchain. You can do this with your online laptop.

1. First, I recommend that you decode the transaction to double check that the values are as you expect. Go to https://blockchain.info/decode-tx and paste in the hex representation of your signed transaction. Double check that the outputs have the addresses and values you expect. Notice that the output values are in Satoshis instead of bitcoin in raw transaction data, so divide by 100,000,000 to get the bitcoin values.
2. When you are happy with the transaction, broadcast it. You can do this at a website like https://blockchain.info/pushtx, or with most wallet software (for Electrum, select tools>load transaction>from text, paste in your text, and then hit the broadcast button).

That’s it! You’ve withdrawn your funds. Look up your multisig address on https://blockchain.info to track the status of your withdrawal.

Indirection transaction when withdrawing?
Lay out options of using coin mixers for deposits / withdrawals. Mention risk (trusted third party) / privacy tradeoff.

Use blockchain.info to get UTXOs for the multisig address

Reserach online to find reasonable fee

1. Boot your quarantined computer into Ubuntu following the same procedure as in Storage Section 3. If your quarantined computer or Ubuntu USB drive have been lost or compromised, then set up a new one.

2. For each of two private keys, convert them from english mnemonics back into the bitcoin private key format. Use the following command to do so:
    a. $ bx mnemonic-to-seed <English private key> | bx ec-new

3. The following value will be used in several commands in this tutorial. Copy it to a text file
    a. "2 [<pubkey 1>] [<pubkey 2>] [<pubkey3>] [<pubkey 4>] [<pubkey 5>] 5 checkmultisig"
    b. You can either move your public keys onto the quarantined computer in one of two ways:
        i. Enter them by hand
        ii. Enter in each of 5 private keys. Then run on each one
            1. $ bx ec-to-public <private key>

4. You will need to look up the transaction ID of the transaction that was used to send coins to the multisig address. You can do this by looking up the multisig address on a public blockchain explorer such as blockchain.info.

5. Decide what your receiving address will be

6. **I M P O R T A N T!!!!!** Decide how much bitcoin you will be sending, denominated in satoshi. One satoshi is one hundredth of a millionth of a bitcoin. This gives you the <satoshi amount> field for the next command. If you put in a bitcoin value like 10 instead of the satoshi value 100000000, you will lose your bitcoin. The following bx command can do the math to transform a bitcoin value into a satoshi value
    a. $ bx btc-to-satoshi <bitcoin amount>

7. Next, you will create a transaction to send. The following steps are drawn from here
    a. $ bx tx-encode -i <transaction ID>:0 -o <receiving address>:<satoshi amount>
    b. We will refer to the output of this command as the <encoded unsigned transaction>. Copy the output to a text file.

8. Sign this transaction with the first private key:

a. $ bx input-sign <private key 1> "2 [<pubkey 1>] [<pubkey 2>] [<pubkey3>] [<pubkey 4>] [<pubkey 5>] 5 checkmultisig" <encoded unsigned transaction>
b. This returns an endorsement string. Copy this to a text file. You'll need it later

9. Do the same thing with a second private key and save the endorsement that is returned to a text file

10. Encode the multisig script and save the output for later use:
    a. bx script-encode "2 [<pubkey 1>] [<pubkey 2>] [<pubkey3>] [<pubkey 4>] [<pubkey 5>] 5 checkmultisig"
    b. We will refer to the output of this as the <encoded redemption script>

11. Create the encoded signed transaction:
    a. $ bx input-set "zero [<endorsement 1>] [<endorsement 2>] [<encoded redemption script>]" <encoded transaction>
    b. Save the return value of this. This is the signed transaction

12. Move the signed transaction off the computer using QR code
    a. $ qrencode "<signed transaction hex>" -o signed_transaction

13. Broadcast it!

Verify transaction on both computers

Functionally identical trasnactions CAN be different because the signature has a nonce
This nonce is theoretically random, but can be used maliciously to leak data (e.g. privkey bits)
We want to make sure the same nonce is used on both computers so it's deterministic
And that the nonce we use is safe (just trust bitcoin core? It generates it deterministically
Based on hash of transaction)

Verify receiving address in transaction is correct

Don't transmit transactions to the online computer via a USB stick (could have malware) -- encode the transaction in a QR code or something

http://www.omgubuntu.co.uk/2011/03/how-to-create-qr-codes-in-ubuntu
Visually verify that the QR code actually encodes the right text

Test decode the transaction on the receiving machine and verify it looks right

https://blockchain.info/decode-tx

Appendix A:
Exceptional Security Measures

This protocol is designed to provide strong protection for almost everyone -- even those storing many millions of dollars.

However, it is *not* designed to provide adequate protection for truly exceptional circumstances, such as a *targeted* attack/surveillance effort (electronic or physical) by a well-resourced criminal organization. This appendix briefly outlines additional measures one might consider if further security were needed above and beyond those in the main security protocol.

We do not recommend considering these measures unless you feel you have a strong need. This list is neither complete nor are the practices cost-effective for almost any circumstances. In addition, implementing these measures incorrectly may decrease security rather than increase it.

Digital software security

- **Verify GnuPG installation:** When downloading a new copy of GnuPG on the setup computer, one would ideally also verify the integrity of the download using the signed checksum. This requires having a pre-existing trusted installation of GnuPG available for verifying the checksum signature.

- **Cross-network checksum sourcing:** Using two different computers on two different networks, obtain all the software checksums from the Internet and verify they are identical, to reduce the risk that the checksums are being compromised by a man-in-the-middle attack.

- **Quarantined checksum verification:** Verify all USB checksums from the quarantined computers to eliminate any risk that software was altered between checksum verification on the Setup Computers and when the USB is used in the quarantined environment.[57]

- **Different quarantined software stacks:** Use different software stacks in each quarantined environment to eliminate the risk that a software bug or vulnerability may generate a flawed key. [58] The new stack should include a non-Linux-derived OS and a different Bitcoin wallet.

- **Dedicated pair of environments for each private key:** Use extra environments such that each environment only touches one key both when generating keys and signing transactions.

[57] The only reason the protocol doesn't currently do this is because the process of verifying the App USB checksums happens as part of Ubuntu's apt-get application, which requires network connectivity. It can be done by hand without apt-get, but it's significantly more involved and so was not included in the protocol.

[58] See the design document for details on why this risk is small enough to justify leaving it unaddressed in the protocol.

Expand the definition of “environment” to include the physical location in which the protocol is executed. This way, compromising one environment will only compromise one key.

- **Manual private key modifications:** Make a small number of random modifications to private keys by hand before using them to generate public keys and addresses.

Side channel security

- **Faraday cage:** Use a Faraday cage to protect against electromagnetic side channels (example). Faraday cages can be self-built or professionally built.

- **No QR codes:** Reading and relaying QR codes to a printer requires a networked device, such as a smartphone, which could potentially receive data from side channels. Instead of using QR codes, copy all redemption scripts and transactions by hand, and keep all nearby smartphones powered off and in Faraday bags through protocol execution.[59]

Hardware security

- **Purchase factory-new Setup Computers:** Don’t use existing computers for your Setup Computers. Purchase them factory-new, and never use them on the same network (to reduce the risk of infection by identical malware).

- **Purchase a factory-new printer:** Printers can have malware, which could conceivably interfere with printing the hardcopy of the protocol document. Use a new printer for printing the protocol document. Choose one without wireless capabilities.

- **Purchase non-recommended equipment:** Don’t purchase any of the suggested equipment linked in this document -- if this protocol achieves widespread adoption, that particular equipment may be targeted for sabotage to undermine the protocol (e.g. loaded dice, malware pre-installed on computers, etc.) Select your own comparable equipment from different manufacturers.

- **Purchase from stores:** Buy all equipment from stores, to reduce the risk it will be tampered with before it is delivered to you. Don’t choose the stores nearest your home or office. Don’t leave the equipment unattended until you are done using it.

- **Improved tamper-evident seals on laptops:** After you are done using the laptop, paint over the hinge joints and cover screws with glitter nail polish and take a picture. The randomness of the glitter is difficult to recreate, so if the laptop is tampered with, you can see it, and know not to use it for future protocol operations.

[59] Note that transcription of redeem scripts and transactions is not only a painstakingly long process, but dangerously vulnerable to human error: any mistakes in the initial transcription & storage of the redemption script will cause all funds to be lost.

- **Destroy quarantined hardware after use:** Even if malware somehow saves sensitive data to permanent media in your quarantined hardware, it can't be stolen if you physically destroy it first.

Paper key security

- **Paper key encryption:** Encrypt the contents of your paper keys using BIP38 to further protect against physical theft.[60]

- **Durable storage medium:** TerraSlate paper is extremely rugged, but you might also consider laminating the paper for additional protection. You'll need a thermal laminator [Amazon] and laminating pouches [Amazon]. An even more durable approach would be to engrave the private keys in metal.

- **High-security vaults:** Store keys in high-security vaults that are more resistant to theft and disaster. (example)

- **Geographically distributed storage:** Store keys in distant cities for resilience against a major disaster that wipes out all keys at once.

- **Multiple fund stores:** Mitigate risk by splitting funds across more than one Bitcoin address, each secured using this protocol, and don't keep printed keys from different store in the same place.

Personal security

- **Unique protocol execution site:** Rather than executing the protocol at your home, office, or anywhere else you frequent, choose a new location (e.g. a hotel) that is unlikely to have compromised or surveillance devices present.

- **Avoid location tracking:** To avoid surveillance (including from adjacent rooms, via side channels like radio waves), take steps to avoid location tracking when executing the protocol. Don't carry a GPS-enabled smartphone with you, don't use credit cards for purchases, etc.

- **Conventional personal security:** Home surveillance systems, bodyguards, etc.

[60] Note that the question of how to securely store the passphrase is non-trivial. It should be unique and hard to guess, which means it is non-trivial to remember. If you are confident you can remember it, storing it only in your own memory will not address estate planning needs. If you record it on paper, you need to make sure those papers are stored securely -- they should not be stored with the keys, and there should be a process for checking on them periodically to make sure they are not lost or damaged.

Appendix B:
Identified Attack Surface & Failure Points

This list describes the attack surface and other failure points for this protocol. We include only attacks and failures limited in scope to specific coins. Attacks and failures related to the Bitcoin ecosystem as a whole (newly discovered cryptographic flaws, critical Bitcoin protocol security or scalability failures, etc.) are not included as most are equally likely to impact the value of all Bitcoins worthless whether or not they are secured with this protocol.

This list assumes no security measures from Appendix A are implemented.

Most attacks require the presence of malware, either in or near the quarantined environment. We'll therefore inventory two layers of the protocol's attack surface:

- Ways in which a malware infection might occur

- Ways in which a critical failure might happen (possibly, but not necessarily, due to a malware infection)

Malware infection vectors

- Software
    - OS/App software has malware (i.e. malicious code) built into official distributions
    - Malware on Setup Computer infects OS/App USB software AND checksum verification produces a false positive
        - Checksum false positives could happen becuase:
            - Malware might interfere with the verification process (or the display of its results).
            - There could be a flaw in the checksum verification software or process.
            - The checksum verification software could be compromised.
                - Verifying the integrity of GnuPG requires one have access to a trusted installation of GnuPG, but many of our protocol users won't have that. The protocol currently recommends users simply trust the version of GnuPG they download.
    - Malware on Setup Computer infects OS/App USB software AFTER checksum verification produces a true positive (i.e. before/during copying of software to the USB, or during USB ejection)

- Firmware
    - USB firmware protection fails AND malware on Setup Computer infects USB firmware
    - Laptop or USB firmware has malware in the shrinkwrapped package

- Hardware
    - Laptop or USB hardware has “malware”[61] in the shrinkwrapped package

Failure scenarios

Electronic failures

- Exfiltration of critical data (e.g. private keys)
    - A Quarantined Computer leaks sensitive data over a side channel (possibly due to malware) AND complementary malware on a (networked or attacker-controlled) device in range steals the data
        - Visual side channel[62] (does not require malware on the quarantined computer, since sensitive data is displayed on the screen as part of the protocol)
        - Acoustic side channel, if inadequately blocked (i.e. insufficient sound blockage or masking noise) (example)
        - Radio side channel (example 1, example 2, example 3)
        - Seismic side channel (example)
        - Thermal side channel (example)
        - Magnetic side channel (example)
    - Malware on a Quarantined Computer exfiltrates sensitive data via QR codes AND cooperating malware on the QR reading device steals the data[63]
- Undetected generation of flawed sensitive data (e.g. easily-guessable private keys, transactions with output addresses belonging to an attacker, etc.)
    - Compatible malware present on BOTH quarantined environments[64]

[61] e.g. a USB JTAG exploit or chip-level backdoors (such as this rootkit). “Malware” usually refers to software, but we’re using it here more broadly to mean “computing technology which undermines the integrity of the computing environment in which it resides.”

[62] If the protocol is followed, the attack surface here should be narrow, as users are instructed to block all visual side channels. However, at a minimum, they are using their smartphone for reading QR codes, and that has a camera on it.

[63] The protocol reduces this risk by putting the QR reader in airplane mode and keeping it there until the QR code is decoded and its content verified manually. But this does not eliminate the risk: malware on the QR reader could override the OS and transmit sensitive data, despite being set to airplane mode. Much more obscurely, malware could construct a false positive during the verification process, by (1) steganographically encoding the sensitive data within the QR code in addition to the “correct” data, and (2) showing *only* the “correct” data when decoding the QR code.

[64] For example, identical “flawed key generation” malware factory-installed on two eternally quarantined laptops from different manufacturers.

Physical failures

- Two paper keys are stolen by an attacker
- All (or all but one) paper keys are lost or destroyed
- An attacker with physical line-of-sight to the laptop takes a photo of the screen while sensitive data is displayed
- Malware on the quarantined machines writes sensitive data to persistent media (USB or laptop hard drive) AND the hardware is physically stolen afterward

Protocol failures

- Protocol hosting (i.e. DNS, Github, website hosting, etc.) is compromised to inject weaknesses into the protocol documentation or software[65]
- Protocol delivery is compromised (e.g. with a man-in-the-middle attack on the user's computer or network) to deliver or display a weakened version of the protocol documentation or software
- Protocol hardcopy is compromised (e.g. by malware to alter the user's hardcopy as it is printed)
- A flaw in ProtocolScript causes sensitive data to be leaked or flawed
- Human error during protocol execution
- Design failure in the protocol misses or inadequately addresses a risk

[65] We mitigate this by signing a checksum of the protocol document itself, and including steps in the protocol for users to verify the signature and checksum. But this is not foolproof:

- An attacker could remove the self-verification procedure from the protocol, and many users would not notice.
- An attacker could compromise our keypair and create a fraudulent signature.
- The protocol document does begin with document self-verification on one Setup Computer. However, it doesn't guide the user through self-verification on the second Setup Computer. Nor does it have them re-verify the document when they first boot into Ubuntu on the Setup Computers to create the Quarantined Boot USBs. If the portion of the protocol document related to creating the Quarantined Boot USBs were compromised between the initial self-validation & the later re-validation (when creating the Quarantined App USBs), the user would probably not notice, even without a forged signature.

Appendix C:
Possible Future Protocol Improvements

No Address Reuse

Currently, this protocol re-uses addresses for both depositing and withdrawing funds. As discussed in the protocol design document, this has both privacy and security implications.

The major hurdle for implementing this is HD wallets, which would allow one to generate one master key, and then use new derived addresses for each deposit or change transaction. Bitcoin Core does not yet support importing user-generated HD wallets in a straightforward way. (details)

Avoiding address re-use would also prevent the use of a test withdrawal. Careful consideration would need to be given as to whether there is another way to safely test funds access, perhaps using something like the signrawtransaction Bitcoin Core RPC.

BIP39 Mnemonic Support

BIP39 supports the creation of private keys encoded as an English mnemonic for ease and reliaiblity of transcription. It's not yet supported by this protocol because it's not supported by Bitcoin Core.

Consider Vanilla Multisig vs. P2SH Transactions

The protocol currently uses P2SH transactions. This allows all signatories storing private keys to view the user's balance, because a copy of the redeem script must be kept with each private key. Vanilla multisig transactions would address this, but it's not clear if it's possible to do vanilla multisig configurations with over 3 keys.[66]

Automate Quarantined USB creation

Many of the steps for creating the Quarantined USBs could be automated in a simple script.

[66] http://bitcoin.stackexchange.com/questions/23893/what-are-the-limits-of-m-and-n-in-m-of-n-multisig-addresses

Appendix D:
Recommended Bitcoin Ecosystem Improvements

This protocol is lengthy and complex because the tools for high-security cold storage do not exist. This appendix briefly outlines some of the tool functionality that would address this gap.

Ideally, the Bitcoin community (and other cryptocurrency communities) will create these tools as soon as possible and render this protocol obsolete. We invite inquiry and consultation by others interested in developing these tools.

Cold Storage Hardware Wallets

- Function like conventional hardware wallets, but eternally quarantined (no wireless *or wired* connections)

- I/O
    - Keyboard for entering data (key recovery, user entropy for key generation)
    - Camera for reading QR codes (for unsigned transactions)
    - Screen for displaying data, including QR codes (for complex data such as signed transactions)

- Generate keys from user-provided entropy (ideally two XOR'd sources)

- Support for BIP39 and HD keys

- Multisig support
    - Each wallet storing one key is probably the way to go
    - Ability to for each device to add one single signature to a transaction, so only one key needs to be stored on a given device
    - Compatibility with HD keys

- Verifiability
    - All deterministic algorithms (for key generation, transaction generation, etc.)
    - Multiple wallet products on the market which use as many different hardware components as possible (to minimize the possibility of a common flaw / vulnerability)

- Simple to use
    - Display steps user through security steps -- how to safely generate their entropy, double-checking that addresses are correct, verifying duplicate algorithm results on an alternate device, etc.

- Optional side channel protection

    - Partner with a company that manufactures some sort of Faraday glove box, and market it to customers who have extra-high security concerns

Bitcoin Core improvements

Until robust cold storage hardware wallets are created, improvements in Bitcoin Core could go a long way towards simplifying this protocol, including reducing the necessary complexity of ProtocolScript.

- Generate keys based on raw user entropy
    - XOR that entropy with /dev/random

- BIP39 key generation support
    - Promotes security through ease of use, and reduces risk of transcription errors

Appendix E:
Release Notes

Version 0.1 Alpha: January 6, 2017

Initial non-public release to selected reviewers.

Appendix F: Contributors

Authors & Maintainers

The protocol was developed and maintained by:

James Hogan (email)
Jacob Lyles (email)

Security Advisors

Our security advisors have offered their substantial time and expertise to consult on the development of this protocol:

Glenn
Enki
Arthur
Greg m?
Philip martin?

Contributors

The following individuals have offered feedback or contributions during the development of this protocol:
Brian Armstrong
Greg Maxwell
Kristov Atlas (http://anonymousbitcoinbook.com/pages/about-the-author)
Andrew C.
Lasse Birk Olesen
Martin Schwarz (https://martinschwarz.wordpress.com/about/)
CubicEarth on bitcointalk.org forums