"If I maintain my silence about my secret it is my prisoner...
if I let it slip from my tongue, I am ITS prisoner."

--- Arthur Schopenhauer ---

©2014, Taryn VanWagner

Secrets:

Obviously the best way to keep a secret is to keep it to yourself, so why use a keychain at all.

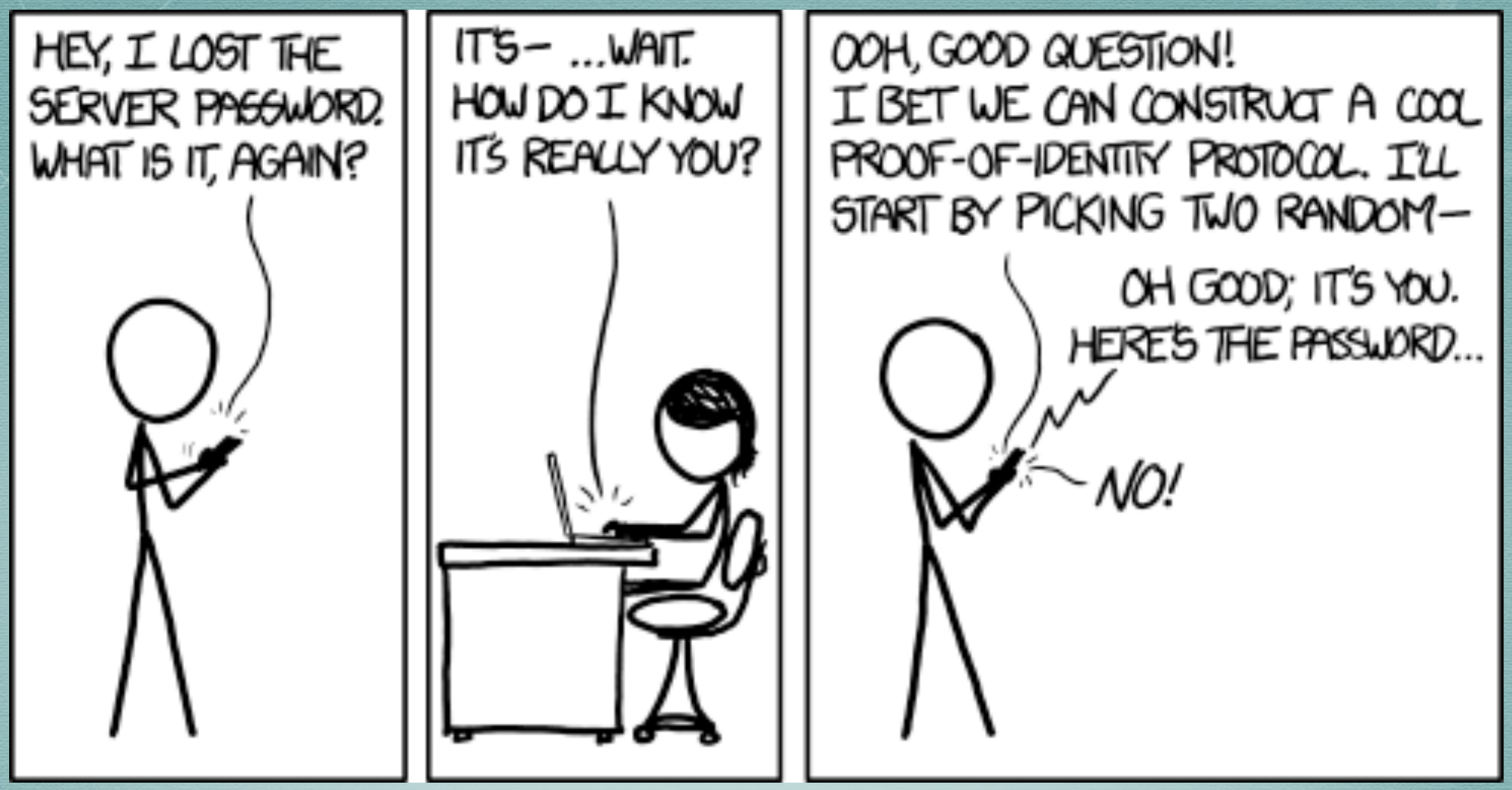
Schopenhauer: 19th century German philosopher, fond of eugenics, possibly an elitist jerk.

"Memory itself is an internal rumour; and when to this hearsay within the mind we add the falsified echoes that reach us from others, we have but a shifting and unseizable basis to build upon. The picture we frame of the past changes continually and grows every day less similar to the original experience which it purports to describe."

--- George Santayana ---

©2014, Taryn VanWagner

George Santayana reminds us that we can't even trust ourselves to properly remember the details of our lives. We need a reliable way to store our secrets or we'll lose our secrets.



Availability, Integrity, Confidentiality, Accountability

Underlying Technical Models for Information Technology Security
NIST Special Publication 800-33

©2014, Taryn VanWagner

In three panels, Cueball & Megan have have touched on all of these security legs:
If you can't recover the password to the server, you have no Availability to the data.
If the password is given to a stranger, Integrity and Confidentiality are lost.
Without proper tests of identity there is no Accountability.

XKCD.com is written by Randall Munroe

The National Institute of Standards and Technology disseminates "Best Practices" on publication 800-33.
There are other "legs" on the security toolbench as well, authors and organizations list extra legs including:
Assurance, which boils down to "Did we do everything else right?"
Also Donald Parker's Security Hexad:
Confidentiality, Control, Integrity, Authenticity, Availability, Utility

Given an item of information:

* Who said it? How do we know they're them? (Accountability)
* Is this really what they said? Is this what they always said? (Integrity)
* Who else knows it? Who else should know it? (Confidentiality)
* Can those who should know it retrieve it when the need it? (Availability)

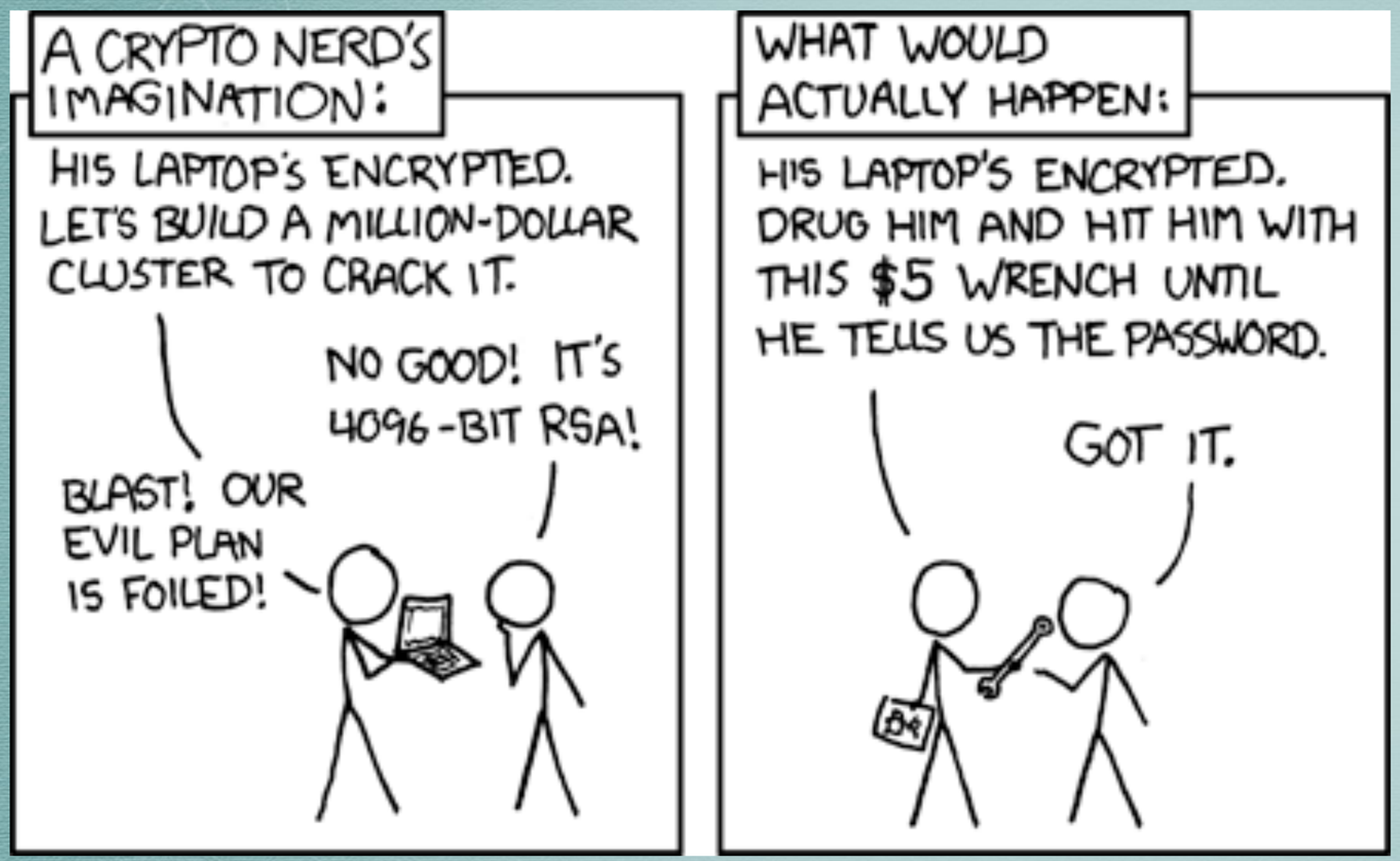
©2014, Taryn VanWagner

In law lingo, the first two items are equivalent to “A chain of evidence.”

The last two are concerned with ownership: “It’s mine, you can’t see it” and “It’s mine so I can always take it out and look at it when I want.”

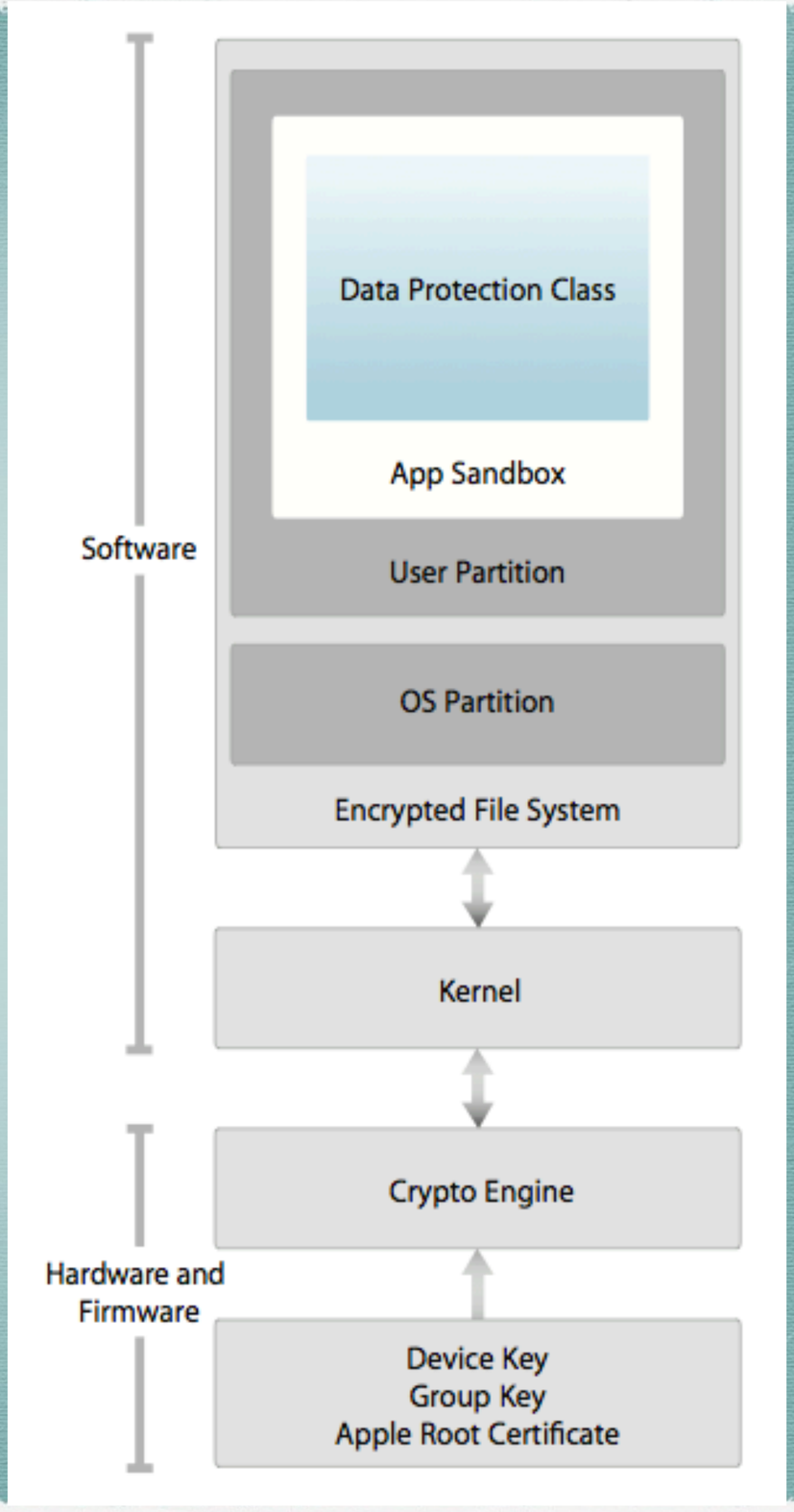
Note that we explicitly do not ask "is it true?", because Information security cannot answer that question.

iOS security



See Apple's "iOS Security, February 2014"
*also see XKCD.com

©2014, Taryn VanWagner



Signed boot process, signed systems and apps.
Sandboxes.
Full system file encryption, with AES 256 hardware support.
App controllable file encryption.
Certificate chains
Isolated security processors, fingerprinting.
SSL v3, TLS, etc.

The Keychain, Keybags

NSUserDefaults plist files are not encrypted (so far?). They can be found in your app's sandbox at Library/Preferences/[bundle_Id].plist

Sandboxes

The sandboxes protect your app from the bad behavior of other apps, by isolating your filesystem from other apps apps.

Now that's a sandbox! This is a Wadi near Sde Boker in the Negev Desert.

File Encryption

Entirely automatic, with useful defaults.

createFileAtPath:contents:attributes:

* NSFileProtectionComplete
* NSFileProtectionCompleteUnlessOpen
* NSFileProtectionCompleteUntilFirstUserAuthentication (default)
* NSFileProtectionNone

©2014, Taryn VanWagner

NSFileProtectionComplete:
On device lock, the decrypted class key is discarded, rendering all data in this class inaccessible until the user unlocks.

NSFileProtectionCompleteUnlessOpen:
For files needed when in background.

NSFileProtectionCompleteUntilFirstUserAuthentication:
Same way as Complete Protection, except that the decrypted class key is not removed from memory when the device is locked. Protects data from attacks that involve a reboot. Default class for all third-party app data not otherwise assigned to a Data Protection class.

NSFileProtectionNone: All the keys needed to decrypt files in this class are stored on the device, the encryption only affords the benefit of fast remote wipe.

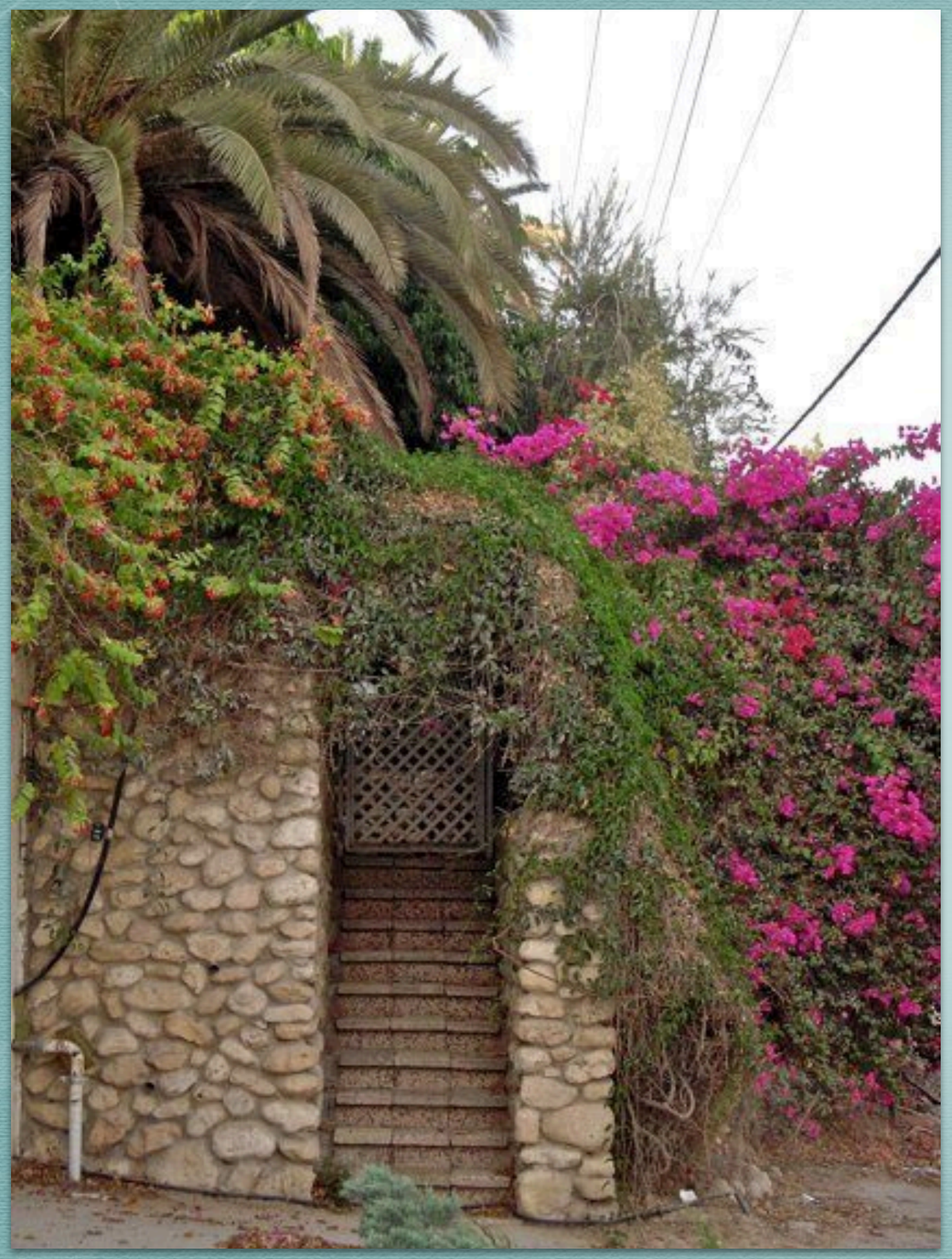
Keychain

* A SQLite database on the system's filesystem
* The securityd daemon gates process and app access to records.
* Data protection classes:

  kSecAttrAccessibleWhenUnlocked
  kSecAttrAccessibleAfterFirstUnlock
  kSecAttrAccessibleAlways

©2014, Taryn VanWagner

The Keychain APIs call through to the daemon, which queries the app's "keychain-access-groups" and the "application-identifier" entitlement.
Data protection classes:



Privacy and Secrets

©2014, Taryn VanWagner

This wonderful entrance is in Gedera, in central Israel. Every time I walked past it I wondered what went on up those stairs and behind those walls.

Addresses, phone numbers, our business correspondence, and the details of our sex lives are not secrets. We must share them with others in order to go about our lives.
Some things end up shared with very few indeed, but still...
On the other hand, these are all data that we are taught should be kept as private as possible, shared only with those we choose to interact with.

One way to gauge how “secret” something should be is to consider the immediacy of ugly consequences if the secret were widely disseminated.
Debit card number and PIN? Login details of a net connected system? Immediate misery, you’ll spend days or weeks cleaning up. Thinking about this, we might all feel a little horrified by the number of times we’ve put our social security number on a form that disappeared into a faceless bureaucracy.

When discussing security, “secret” has a more constrained sense than simply meaning private data. Secrets are the keys used to unlock systems, to validate identity, and to authenticate the data they guard. Secrets must be held much more closely than mere private data.

iOS provides APIs for securing network traffic, for encrypting application files, and for keeping secrets.

If an application collects user data it is may be best kept in an encrypted file, NOT in a preferences files. These are fast to use, and the operating system transparently manages locking and unlocking the file as needed.

But the keys to decrypt those files, and the user’s login to your app’s network services, and the credit card numbers the user has asked you to keep, all need to be kept even more securely, more secretly, and encrypting and managing them is the keychain’s purpose.

How do Keychains and Encryption Support Security Models?

©2014, Taryn VanWagner

In isolation, encrypting a secret does not guarantee that the secret came from a trusted source. Accountability in modern communications is provided by certificates, and their capacity to establish a chain of trust via public and private keys. Note that a certificate is only as good as the issuer's ability to keep their private keys secret, so the presence of a valid certificate is not an absolute guarantee.

Nor does encryption alone provide Integrity, we cannot know if the secret was modified while traveling to us, unless we have accountability through the use of certificate based channels.

All we get from the encryption and the keychain is confidentiality and availability.

Keychain API

Talking to the daemon

©2014, Taryn VanWagner

I was trying for ominous, how did I do?

All the photos in this presentation are mine, taken in florida and the middle east.

Keychain calls

```
#import <Security/Security.h>

OSStatus SecItemAdd(          CFDictionaryRef attributes,
                              CFTypeRef *     result)

OSStatus SecItemUpdate(       CFDictionaryRef query,
                              CFDictionaryRef attributesToUpdate)

OSStatus SecItemCopyMatching(CFDictionaryRef query,
                              CFTypeRef *     result)

OSStatus SecItemDelete(       CFDictionaryRef query)
```

©2014, Taryn VanWagner

The first annoying bit is that they are Core Foundation functions instead of an ObjectiveC or Swift class, how could Apple be so hopelessly backwards!

Their arguments and return values don't seem particularly intuitive either:

```
typedef SInt32                          OSStatus;
typedef const struct __CFDictionary *   CFDictionaryRef;
typedef const void *                    CFTypeRef;
```

As absurd as it might seem for an argument which is updating or deleting an element to be called "query", remembering that this is actually an API to an SQL database gives an historical context for the silly name. Then it just becomes silly that in one case, they call it "attributes" instead of "query"

It's also not apparent why adding an item requires both status return and a result argument.

And of course, we've few clues on what sorts of types the results might be.

Maybe the function errors returns will give some hints?

Keychain error returns

Result Code	Value	Description
errSecSuccess	0	No error. Available in iOS 2.0 and later.
errSecUnimplemented	-4	Function or operation not implemented. Available in iOS 2.0 and later.
errSecParam	-50	One or more parameters passed to the function were not valid. Available in iOS 2.0 and later.
errSecAllocate	-108	Failed to allocate memory. Available in iOS 2.0 and later.
errSecNotAvailable	-25291	No trust results are available. Available in iOS 2.0 and later.
errSecAuthFailed	-25293	Authorization/Authentication failed. Available in iOS 4.2 and later.
errSecDuplicateItem	-25299	The item already exists. Available in iOS 2.0 and later.
errSecItemNotFound	-25300	The item cannot be found. Available in iOS 2.0 and later.
errSecInteractionNotAllowed	-25308	Interaction with the Security Server is not allowed. Available in iOS 2.0 and later.
errSecDecode	-26275	Unable to decode the provided data. Available in iOS 2.0 and later.

©2014, Taryn VanWagner

Duplicate item and item not found are suggestive, telling us why we have both add and update. We won't be allowed to add an existing item or update a non-existing item.
Update uses a query parameter because it will be finding the element first.

But these error returns imply that "add" will also be doing a query, but it will fashion the query from the attributes parameter.

Keychain “query” dictionaries

“A typical search dictionary consists of:

- The class key-value pair, which specifies the class of items (for example, Internet passwords or cryptographic keys) for which to search.
- One or more key-value pairs that specify the attribute data (such as label or creation date) to be matched.
- One or more search key-value pairs, which specify values that further refine the search, such as issuing certificates or email addresses to match.
- A return-type key-value pair, specifying the type of results you want (for example, a dictionary or a persistent reference).

Which attributes can be specified depends on the class of the item for which you wish to search.”

From Apple’s “Keychain Services Programming Guide”

©2014, Taryn VanWagner

Notable things:
So far we are left to guess at which combinations are usable.
What’s up with this idea of the return type?

Keychain Class Constants

```
// From SecItem.h (most comments are mine)
CFTypeRef kSecClass;  // dictionary key marking a class type entry
// Possible class types:
CFTypeRef kSecClassGenericPassword;  // Generic password (a string)
CFTypeRef kSecClassInternetPassword; // Adds server, protocol, etc.
CFTypeRef kSecClassCertificate;      // Not entire cert?
CFTypeRef kSecClassKey;              // Crypto keys.
CFTypeRef kSecClassIdentity;         // Cross of Certificate and Key
```

©2014, Taryn VanWagner

So the keychain stores five different classes of secrets, and uses the same APIs for all of them. The parameter dictionaries include class types.

I have more questions than answers regarding these variants. Each has a large selection of required and optional attributes.

In fact, the ONLY attribute required in all the calls is the class of the desired record. I'd guess that this implies that the underlying implementation uses separate database tables for each class.

Attributes common to all keychain classes

kSecClass

```
 kSecClassGenericPassword
kSecClassInternetPassword;
kSecClassCertificate
kSecClassKey
kSecClassIdentity
```

kSecAttrAccessGroup

```
String identifying the apps or
services permitted to access the
group, e.g. com.ornae.secrets
```

kSecAttrAccessible...

```
Enumerated constants
  kSecAttrAccessibleAlways
  ...WhenUnlocked
  ...AfterFirstUnlock
  ...WhenUnlockedThisDeviceOnly
  ...AfterFirstUnlockThisDeviceOnly
  ...AlwaysThisDeviceOnly
```

kSecAttrLabel

```
Human readable label.
A string for labeling this item.
(Perhaps it might be used to label
a text entry field?)
```

©2014, Taryn VanWagner

Looking at kSecAttrAccessGroup, there's a hint of a possible security flaw. Could someone else create an app that impersonates a bundleID like com.ornae.secrets and surf through the keychain entries of Ornae's secrets? My suspicion is not, because underlying permissions are derived from developer certificates.

If any of you have access to several different developer accounts (and lots of spare time) you might like to play with this. The simplest defense against such an attack is to not create an access group attribute, there's no point using it unless you're delivering a suite of apps.

kSecAttrLabel might possibly be used as a universal key for all variants of keychain classes. None of the other common attributes presents itself as a candidate. OTOH, Apple's example code doesn't use it like that.

GenericPassword & InternetPassword Common Attributes

kSecAttrCreationDate
Uses the CFDateRef type

kSecAttrModificationDate
Uses the CFDateRef type

kSecAttrDescription
Human readable label.

kSecAttrComment
“User editable comment”?

kSecAttrCreator
OS9‘s 32 bit creator code

kSecAttrType
OS9‘s 32 bit file type

kSecAttrIsInvisible
CFBooleanRef, displayable?

kSecAttrIsNegative
CFBooleanRef, retainable?

kSecAttrAccount
Possibly a database key

©2014, Taryn VanWagner

kSecAttrDescription seems to overlap the uses for kSecAttrLabel, but is probably not a database key, even though matchable.

Look at the first two in the right column. Yes, the item's creator and type attributes. “...use this key to set or get a value of type CFNumberRef that represents the item's creator. This number is the unsigned integer representation of a four-character code (e.g., 'aCrt').”

kSecAttrIsNegative has an unfortunate name. In a better world it would have a name like kSecAttrIsPasswordRetainedInKeychain ... but noooo!!! What’s more, I’ve read the descriptions a few times, and I’m still not sure which way the logic is supposed to operate.

GenericPassword Unique Attributes

```
kSecAttrGeneric
kSecAttrService
```

InternetPassword Unique Attributes

```
kSecAttrSecurityDomain
kSecAttrServer
kSecAttrProtocol
kSecAttrAuthenticationType
kSecAttrPort
kSecAttrPath
```

SecItem.h has the best description of all the keychain constants.

©2014, Taryn VanWagner

SecItem.h is really useful, I'd never have made sense of this with just the docs.

Attributes for kSecClassCertificate, kSecClassKey, & kSecClassIdentity

```
kSecAttrCertificateType
kSecAttrCertificateEncoding
kSecAttrSubject
kSecAttrIssuer
kSecAttrSerialNumber
kSecAttrSubjectKeyID
kSecAttrPublicKeyHash

kSecAttrKeyClass
kSecAttrApplicationLabel
```

```
kSecAttrIsPermanent
kSecAttrApplicationTag
kSecAttrKeyType
kSecAttrKeySizeInBits
kSecAttrEffectiveKeySize
kSecAttrCanEncrypt
kSecAttrCanDecrypt
kSecAttrCanDerive
kSecAttrCanSign
kSecAttrCanVerify
kSecAttrCanWrap
kSecAttrCanUnwrap
```

©2014, Taryn VanWagner

I'm showing you these, but they are beyond the scope of tonight's discussion. Tonight I'm only focusing on storing the simplest of secrets, things like credit cards, debit cards, banking pins etc.

As we've looked at all these constants associated with securing data, they make sense individually, but it all seems way more complicated than it needs to be.

So, why is this all so obtuse?

The short answer seems to be CDSA, Intel's Common Data Security Architecture (a design)
and CSSM, the Common Security Services Manager (an API and library)
These were adopted by the Open Group, including Apple, and implemented as part of Darwin.
I believe the keychain is built on that implementation. CSSA was deprecated with OS 10.7, and was never a public API in iOS, but the keychain's SecItem functions have a stench of being designed by an enterprise sponsored committee.

Apple's KeychainItemWrapper demo

```
/*  The KeychainItemWrapper class is an abstraction layer for the iPhone Keychain
  communication. It is merely a simple wrapper to provide a distinct barrier between
  all the idiosyncracies involved with the Keychain CF/NS container objects. */
@interface KeychainItemWrapper : NSObject
{
    NSMutableDictionary *keychainItemData;   // The actual keychain item data backing store.
    NSMutableDictionary *genericPasswordQuery; // A placeholder for the generic keychain item
query used to locate the item.
}

@property (nonatomic, retain) NSMutableDictionary *keychainItemData;
@property (nonatomic, retain) NSMutableDictionary *genericPasswordQuery;

// Designated initializer.
- (id)initWithIdentifier: (NSString *)identifier accessGroup:(NSString *) accessGroup;
- (void)setObject:(id)inObject forKey:(id)key;
- (id)objectForKey:(id)key;

// Initializes and resets the default generic keychain item data.
- (void)resetKeychainItem;

@end
```

©2014, Taryn VanWagner

As an abstraction layer I think this is a fail.
Along with the backing code not being ARC compliant, I don't think this API provides a very good "barrier between all the idiosyncracies involved with the Keychain CF/NS container objects"

After all what do we want from a secret storage API?

An API for storing secrets

```
// Retrieve a value from the keychain
NSString* valueForKey:(NSString *)key;

// Store value in the keychain using key
(bool) storeValue:(NSString *)value forKey:(NSString *)key;

// Completely remove entry from keychain
(bool) removeAllForKey:(NSString *)key;
```

©2014, Taryn VanWagner

Granted, this does not support access groups, but supporting access groups violates the 80% rule.

The KeyRing API

```
//  KeyRing.h
//  Created by taryn on 6/22/14.
//  Copyright 2014. All rights reserved.
//
//  Simple wrapper to store key and string pairs in the keyChain.
//  NOT to be used for https traffic etc, there are better ways in iOS.

//  No instance methods because there was no reason to create instances.
//  The only reason it’s a class at all was to use named parameters

#import <Foundation/Foundation.h>

@interface KeyRing : NSObject

// Retrieve a value from the keychain
+(NSString*) valueForKey:(NSString *)key;

// Store value in the keychain using key
+(bool) storeValue:(NSString *)value forKey:(NSString *)key;

// Completely remove entry from keychain
+(bool) removeAllForKey:(NSString *)key;

@end
```

©2014, Taryn VanWagner

Keyring: Build a Query, Remove an Entry

```
+(NSMutableDictionary*) newQueryDictionaryforKey:(NSString *)key {
    NSMutableDictionary *query = [NSMutableDictionary dictionaryWithObjectsAndKeys:
                        (__bridge id)kSecClassGenericPassword, kSecClass, // object is a string
                        key,                                   kSecAttrAccount, // identified by key
                        kCFBooleanTrue,              kSecReturnAttributes, // decrypt to dictionary
                        nil];
    return query;
}


// Completely remove entry from keychain
+(bool) removeAllForKey:(NSString *)key {

    NSDictionary *query =  [KeyRing newQueryDictionaryforKey:key];
    OSStatus status = SecItemDelete( (__bridge CFDictionaryRef) query);
    if (status == noErr) {    // Value found?
        return true;          // found it, good exit
    }
    NSLog(@"%s No value found for key:%@, error is %i",__func__, key, (int)status);
    return false;
}
```

©2014, Taryn VanWagner

The code on this and the next few slides is obsolete, please use the project at https://github.com/taryntoo/Secrets-the-KeyRing

Note the cast on kSecClassGenericPassword, it's needed because that isn't really a constant. It's a pointer to an enumeration cast as a void * ... I think.

Keyring: Fetch a password

```
//  query = [NSMutableDictionary dictionaryWithObjectsAndKeys:
//                ...
//       kCFBooleanTrue,  kSecReturnAttributes, // // decrypt to dictionary
//                ...

// Retrieve a value from the keychain
+(NSString*) valueForKey:(NSString *)key {

    CFTypeRef result; // NSDictionary *result;
    NSMutableDictionary* query = [KeyRing newQueryDictionaryforKey:key];

    OSStatus status = SecItemCopyMatching( (__bridge CFDictionaryRef) query,
                                           (CFTypeRef *)             &result);

    if (status == noErr) {    // Value found?
        NSString *value = (NSString *) [(__bridge NSDictionary*) result
                        objectForKey: (__bridge NSString *)    kSecAttrGeneric];
        return value;   // found it, good exit
    }
    NSLog(@"%s No value found for key:%@, error is %i",__func__, key, (int)status);
    return nil;  // Not found
}
```

©2014, Taryn VanWagner

The code on this page includes a memory leak and has since been substantially rewritten, and fixed. Please use the project at https://github.com/taryntoo/Secrets-the-KeyRing

CFTypeRef result is a void * so how did I know that this is really a dictionary?
Docs tell us that:
query dictionaries should include “A return-type key-value pair, specifying the type of results you want (for example, a dictionary or a persistent reference).”

Have a look up here at the query dictionary.
kSecReturnAttributes
Return attributes attribute key.
The corresponding value is of type CFBooleanRef. A value of kCFBooleanTrue indicates that a dictionary of the (unencrypted) attributes of an item should be returned in the form of a CFDictionaryRef.

Keyring: set a password

```
// Store value in the keychain using key
+(bool) storeValue:(NSString *)value forKey:(NSString *)key {

    OSStatus status = 0;
    NSMutableDictionary* query = [KeyRing newQueryDictionaryforKey:key];

    NSString *existingValue = [self valueForKey:key];   // Does the value exist already?
    if (existingValue) {                                // Yes, so we'll be doing an update
        NSDictionary *valueDict = [NSDictionary dictionaryWithObjectsAndKeys:
                                   value, kSecAttrGeneric, nil]; // the value we're replacing or adding.

        // Keychain complains if this attribute appears in an update
        [query removeObjectForKey:(__bridge id<NSCopying>)(kSecReturnAttributes)];

        status = SecItemUpdate( (__bridge CFDictionaryRef)query,
                                (__bridge CFDictionaryRef)(valueDict) );
    } else {
        // existing value not found, we'll need it in the dictionary we're passing to SecItemAdd()
        [query setObject:value forKey:(__bridge id<NSCopying>)(kSecAttrGeneric)];

        status = SecItemAdd((__bridge CFDictionaryRef)(query), NULL); // we can ignore any result param.
    }

    if (status == noErr) {  // was the value stored?
        return true;        // yes, good exit
    }
    NSLog(@"%s failed, error is %i",__func__, (int)status);
    return false;   // something failed.
}
```

©2014, Taryn VanWagner

The code on this page is obsolete, please use the project at https://github.com/taryntoo/Secrets-the-KeyRing

I'm not sure if what I've done here is good practice. It's not how you'd do it in SQL, if I recall correctly. Note that StoreValue provides both Add and Update as a single method.

We check if the value is there first, we're forced to by the underlying implementation.

Don't forget that if the calling application is concerned with not overwriting existing passwords, it can just check using valueForKey itself, and that will entirely decouple our logic for writing with the application's logic for deciding whether to write.

A call to SecItemUpdate needs both a query dictionary and a values dictionary, this also implies a database mentality: select a record, act on a record.

And there's also a requirement to strip the kSecReturnAttributes value out of the query dictionary, even though this code actually works if i leave it in the dictionary, a "bad parameter" error is returned.

So, questions before we try running the code?

off to xcode.....