Breaking into the OS X keychain

***Update**: I want to clear up some misconceptions. This is not a security bug in OS X. Everything works as designed. The point of this post was to show a post-exploitation technique and to release a tool for the job. I found this particular technique interesting because it is instantaneous, reliable across OS X versions, and requires no persistent changes in the system.*

TL;DR: Root can read plaintext keychain passwords of logged-in users in OS X. Open source proof-of-concept.

There is a design compromise in Apple's keychain implementation that sacrifices some security for a lot of usability.

As a result, the root user is able to read all keychain secrets of logged-in users, unless they take extra steps to protect themselves. I'm sure Apple is perfectly aware of the security implications, and made the bargain intentionally.

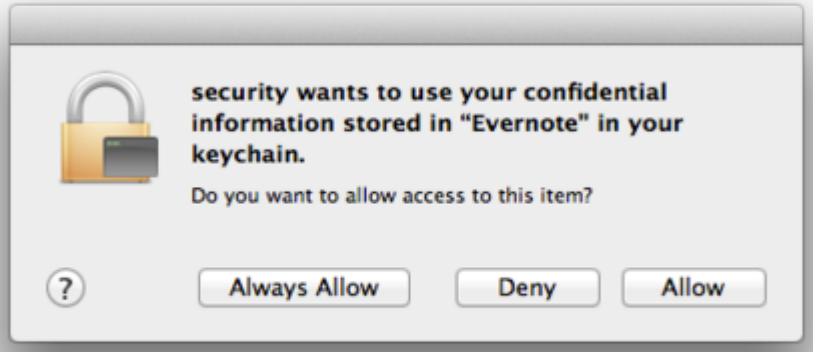
Because this is an intentional design decision instead of a security bug, its exploitation should not come as a surprise. However, I haven't seen anyone actually use or mention any practical methods before in public.

The situation

In OS X, your keychain contains your saved passwords. This includes all your email accounts in Mail, passwords stored in Safari, and credentials for accessing known Wi-Fi networks. Because it contains valuable secrets, the keychain is encrypted. It can only be opened with your login password.

But there's a twist. When you log in to OS X, the operating system automatically unlocks your keychain for your convenience. This means that you don't have to enter your login password every time you want to use your stored passwords.

That's why, by default, you see keychain dialogs like this



instead of this

Notice that only one of them asks for your password. Most users are probably much happier with the no-password-needed dialog, so that's what Apple implemented. Of course, this means that it has to be somehow possible to read your keychain passwords even without asking for your login password every time. That's what "unlocking" means here.

So, your passwords are encrypted to keep them safe, but they have to be readable by OS X itself to keep you happy. That's a bit of a dilemma. Apple's solution is threefold.

- Unlocking does not change the keychain file. The plaintext passwords do not appear on disk (or in memory), even when they are unlocked.
- Apple provides an official API for reading unlocked passwords. Even though it *does* allow you to read them, it also asks for the user's permission with a dialog window.

- What unlocking actually does or how the unlocked passwords are accessed is not documented. This is security through lack-of-documentation.

The attack

The passwords in a keychain file are encrypted many times over with various different keys. Some of these keys are encrypted using other keys stored in the same file, in a russian-doll fashion. The key that can open the outermost doll and kickstart the whole decryption cascade is derived from the user's login password using PBKDF2. I'll call this key the *master key*.

If anyone wants to read the contents of a keychain file, they have to know either the user's login password or the master key. This means that `securityd`, the process that handles keychain operations in OS X, has to know at least one of these secrets.

I put this observation to the test on my own laptop.

Scanning `securityd`'s whole memory space did not reveal any copies of my login password. Next, I used PBKDF2 with my login password to get my 24-byte master key. Scanning the memory again, a perfect copy of the master key was found in `securityd`'s heap.

This lead to the next question. If the master keys are indeed stored in `securityd`'s memory, is there a good way to find them? Testing every possible 24-byte sequence of the memory space is not very elegant.

Instead of fully inspecting `securityd`'s whole memory space, possible master keys can be pinpointed with a couple of identifying features. They are stored in `securityd`'s heap in an area flagged as MALLOC_TINY by `vmmap`. In the same area of memory, there's also always structure pointing to the master key. The structure contains an 8-byte size field with the value of 0x18 (24 in hex) and a pointer to the actual data.

The search is rather simple:

- Find `securityd`'s MALLOC_TINY heap areas with `vmmap`
- Search each found area for occurrences of 0x0000000000000018
- If the next 8-byte value is a pointer to the current heap area, treat the pointed-to data as a possible master key

With this kind of pattern recognition, the number of possible master keys is reduced to about 20. Each of the candidates can be used to try to decrypt the next key (which I call the *wrapping key*). Only the real master key should spit out a sensible value for the wrapping key. It's not a foolproof method, but with the low number of candidates it seems to be good enough. I haven't run into a single false positive yet.

The rest of the decryption process is rather tedious. The master key reveals the wrapping key. A hardcoded obfuscation key reveals the encrypted credential key. The wrapping key reveals the credential key. The credential key finally reveals the plaintext password. All glory to Matt Johnston for his research on these decryption steps.

Here is a sample run and its truncated output, with actual passwords and usernames replaced with x's.

```
 $ sudo ./keychaindump
[*] Searching process 15 heap range 0x7fa809400000-0x7fa809500000
[*] Searching process 15 heap range 0x7fa809500000-0x7fa809600000
[*] Searching process 15 heap range 0x7fa809600000-0x7fa809700000
[*] Searching process 15 heap range 0x7fa80a900000-0x7fa80ac00000
[*] Found 17 master key candidates
[*] Trying to decrypt wrapping key in /Users/juusosalonen/Library/Keychains/login.keychain
[*] Trying master key candidate: b49ad51a672bd4be55a4eb4efdb90b242a5f262ba80a95df
[*] Trying master key candidate: 22b8aa80fa0700605f53994940fcfe9acc44eb1f4587f1ac
[*] Trying master key candidate: 1d7aa80fa0700f002005043210074b877579996d09b70000
[*] Trying master key candidate: a0a20000000200f7474d400000700d01a980fa00007f085e
[*] Trying master key candidate: 180000000000000000000000000000000000000007000001
[*] Trying master key candidate: 0000b107000001000000803e970aa8710ae567eff7ff0000
[*] Trying master key candidate: 796a63507e6c2a84d9f095fae2896058dfe029cd0f7105da
[*] Trying master key candidate: 16ac866d636215c01e337e942f48cfed12d7c45bfab8dbf7
[*] Trying master key candidate: 070020539baab0d1d6a3aa80fa006877ed57f80fa0000000
[*] Trying master key candidate: 88edbaf22819a8eeb8e9b75120c0775de8a4d7da842d4a4a
[+] Found master key: 88edbaf22819a8eeb8e9b75120c0775de8a4d7da842d4a4a
[+] Found wrapping key: e9acc39947f1996df940fceb1f458ac74b877579f54409b7
xxxxxxx:192.168.1.1:xxxxxxx
xxxxxxx:news.ycombinator.com:xxxxxxx
xxxxxxx@gmail.com:login.facebook.com:xxxxxxx
xxxxxxx@gmail.com:smtp.google.com:xxxxxxx
xxxxxxx@gmail.com:imap.google.com:xxxxxxx
xxxxxxx@gmail.com:api.heroku.com:xxxxxxx
xxxxxxx:www.freechess.org:xxxxxxx
xxxxxxx:twitter.com:xxxxxxx
xxxxxxx@gmail.com:www.google.com:xxxxxxx
xxxxxxx:imap.gmail.com:xxxxxxx
...
```

You can find the source code for keychaindump at GitHub. It's a proof-of-concept, far from perfect, and it doesn't list all the possible keychain secrets. But it seems to work okay. I've tested it on OS X Lion and Mountain Lion.
