Pico: No more passwords!

Frank Stajano

University of Cambridge Computer Laboratory

Abstract. From a usability viewpoint, passwords and PINs have reached the end of their useful life. Even though they are convenient for implementers, for users they are increasingly unmanageable. The demands placed on users (passwords that are unguessable, all different and never written down) are no longer reasonable now that each person has to manage dozens of passwords. Yet we can't abandon them until we come up with an alternative method of user authentication that is both usable and secure.

We present an alternative design based on a hardware token called Pico that relieves the user from having to remember passwords and PINs. We discuss how Pico would work, not merely in the relatively straightforward case of remote login to a web site but also in all the other contexts in which users must at present remember passwords and PINs. After discussing an ideal clean-slate design we consider possible variations that might trade off some security in exchange for backwards compatibility, convenience and better prospects for widespread adoption.

1 Why users are right to be fed up

Remembering an unguessable and un-brute-force-able password was a manageable task twenty years ago, when each of us had to use only one or two. Since then, though, two trends in computing have made this endeavour much harder. First, the power of computers has grown by several orders of magnitude: once upon a time, eight characters were considered safe from brute force[1]; nowadays, passwords that are truly safe from brute force[2] and from advanced guessing attacks[3] typically exceed the ability of ordinary users to remember them [ref Schneier "Key length and Security" in Crypto-gram 1999-10]. Second, and most important, the number of computer-based services with which we interact has grown out of proportion: we no longer have just one login password to remember but those of several email accounts, web sites and various online services, not to mention a variety of PINs for our bank cards, phones, burglar alarms and so on.

Let's henceforth refer to such password-requesting applications and services as "apps". For the implementer of a new app (say a new web site), passwords are the easiest and cheapest way of authenticating the user. And, since the password

[1] Indeed, systems often didn't even *allow* you to have a longer password.

[2] Thus `long&COmpleXandFullOf$pecialCHars`.

[3] Thus semantically meaningless, as in `u4Hs9DB66Ab18GIUdCVi`.

Preliminary submission to the Security Protocols Workshop 2011
Revision 15 of 2011-03-09 15:02:35 +0000 (Wed, 09 Mar 2011)

method is universally understood and used everywhere else, there is no apparent reason not to continue to use it[4].

This approach doesn't scale to the tens of passwords and PINs each of us must use today. Users frequently complain that they can't stand passwords any longer. And they are right: the requests of computer security people have indeed become unreasonable. Users are told never to write down their passwords but they know they will face substantial hassle, if not total loss of their data or other assets, if they ever forget them: availability is often worth much more to them than confidentiality. If they comply with the request not to write down their passwords, they'll want to choose something memorable. But, they're told, it must also be hard to guess[5]. Once they find a strong, memorable yet hard to guess password, the annoying computer security people also dictate that they can't reuse it anywhere else. It's hard to find twenty or thirty different ones that are all memorable and yet not guessable by someone who knows a little about the user. And those annoying computer people may even request that they *change* them from time to time. Fat chance of remembering them all, then! Not to mention the hassle of having to type them correctly despite the absence of visual feedback ("*********"). Users are justified in being fed up[6].

Imagine we could afford to start again from scratch, and that our primary directive were to invent a user authentication method that no longer relied on users having to remember unguessable secrets. The new method would have to do better (or at least no worse) than passwords in at least the following realms:

Usability: Users should not have to memorize secrets.

Security: The alternative should be at least secure as passwords would be if users were able to follow all those implausible and contradictory rules (memorable, unguessable, never reused etc.).

Scalability: The effectiveness of the method should not be degraded even if the user had to authenticate to dozens or even thousands of different apps.

The design proposed in this paper meets these requirements. It is based on a token that remembers the user's authentication credentials. Since the method is token-based, it should also address these additional concerns:

Availability: If the token is lost or destroyed, the user must be able to regain access to the services.

Confidentiality: If the token is stolen, the thief must not be able to impersonate the user—even assuming that the thief can tamper with the token and access its insides.

[4] It's a classic *tragedy of the commons*: "What harm could my site cause by requesting a password, since everyone else's also does?" And yet, since everyone does, the burden placed on users becomes unmanageable.

[5] Despite the easily retrieved digital footprints we all leave behind online.

[6] And it is obvious that, if the rules of the computer security people are in practice mutually incompatible, they won't all be followed, even when users are genuinely cooperative and willing to comply.

The proposed design also meets these additional requirements. Its main disadvantage, aside from the obvious fact that the user must carry the token in order to authenticate, is that it requires changes to the verifying app in order to offer all its features.

For this reason, after describing the clean-slate design, I revisit it to evaluate a number of possible variations. It may be more advantageous to make the system slightly less secure[7] if this increases convenience, backwards compatibility or market acceptance.

2 Pico: a usable and secure memory prosthesis

The user has a trustworthy device called a Pico[8] that acts as a memory prosthesis and takes on the burden of remembering authentication credentials, transforming them from "something you know" to "something you have". This solves the first and most important problem:

- **Benefit 1**: The user no longer has to remember passwords.

But we could have trivially solved this simply by writing the passwords down on a piece of paper; this, however, offers inadequate confidentiality. A PDA holding an encrypted "password wallet" program such as Schneier's "Password Safe" [ref] comes closer to the idea of the Pico. In section 4.1 we explain how the Pico protects the confidentiality of the credentials it remembers.

If we let the Pico, rather than the user, choose a new strong random password for each account, we solve two additional problems:

- **Benefit 2**: It is not possible for a user to choose a weak (brute-forceable or guessable) password.

- **Benefit 3**: It is not possible for a user to reuse the same credential with different verifiers. Therefore it is not possible for a malicious verifier to impersonate the user elsewhere, or for a careless verifier to cause the user to be impersonated elsewhere.

Note however that, if it must work with existing apps, the Pico has some constraints in the choice of password: as demonstrated by the evidence collected by Bonneau and Preibusch [ref], a significant fraction of web sites will not even *accept* a really strong password[9]. The clean-slate solution would be to force all apps to accept a sufficiently long strong password in a standard format (e.g.

[7] While still more secure than passwords as they are used today.

[8] After Giovanni Pico della Mirandola, an Italian 15^{th} century philosopher known for his prodigious memory. Totally unrelated to the 10^{-12} SI prefix.

[9] Of the 150 web sites surveyed by Bonneau and Preibusch, 38% imposed arbitrary limits on the maximum length of the password. (The authors insightfully take as a clue that those sites probably don't hash passwords before storing them.)

a 128-bit random string in base64 encoding). The pragmatic and backwards-compatible solution would be to tell the Pico, at account creation, the constraints on the password that the app imposes (e.g. between 6 and 10 characters with at least one letter and a number).

Existing systems based on a PDA with a password wallet have the usability problem that, even though you are no longer forced to remember that the password was `u4Hs9DB66Ab18GIUdCVi`, it is still rather tedious and error-prone to have to transcribe it from the PDA to the keyboard of your computer (the more secure the password, the harder it is to type it correctly). The clean-slate solution is thus to allow the Pico to "type" into the app directly, via radio. This solves another problem:

– **Benefit 4**: The user no longer has to *type* the damn password.

An unrelated problem is that a malicious app could masquerade as the genuine app and trick the user into revealing the password. With web sites this is commonly indicated as *phishing*, but a similar attack could also occur locally, for example with a Trojan application displaying a fake login screen on your computer. To prevent this, in the clean-slate design the Pico authenticates the app before supplying the user's credentials, thus offering...

– **Benefit 5**: It is not possible for a malicious app to steal the user's credentials by impersonating the app to which the user wanted to authenticate.

This, however, requires updating the app itself. SSL-equipped web sites already provide most of the necessary machinery; for other apps, particularly local ones, things are more complicated. But solving the password problem only for web sites would not be satisfactory for users: for them, if passwords and PINs are a nuisance, they are a nuisance everywhere, not just on the web. We must provide a holistic solution, even though it will require changes to the verifier side. We discuss the details in sections 3.3–3.7, yielding

– **Benefit 6**: We don't just get rid of passwords for remote web sites but for any other kind of app that wants a password or PIN, including local ones[10] and even ones involving standalone devices[11].

To protect against phishing, some systems such as the one by Parno, Kuo and Perrig [ref] request that the user select the desired web site on the trusted device first, in order to visit the correct app at some well-known network address. This is a very sound principle. However it is hard to enforce when the device is not able to "invoke" the app, as happens in most cases other than web sites[12]: your laptop asks you to unlock it with your login credentials, or a program requires you to

[10] Such as logging into your computer.

[11] Such as unlocking your car stereo or burglar alarm.

[12] It may also be the case that the user is already interacting with a web site app and suddenly further authentication is requested, for example because the user requests a "more sensitive" function such as "show my private data" or "add a new payee".

type the root password. In such cases it's arguably the verifier that initiates the transaction and it's up to the prover to select the appropriate credentials (besides the anti-phishing meta-problem of deciding whether to respond at all). Selecting the credentials from a list will be tedious if there are more than a couple dozen. A hierarchically arranged list will help, but to scale to potentially thousands of apps it's best to let the Pico recognize the app and select the appropriate credentials (or say that it's an unknown and probably malicious app). Thus:

– **Benefit 7**: The user no longer has to manually select the appropriate credentials for the app[13].

A rarely discussed problem of traditional password-based authentication is that, if a session lasts several hours, the app has no way of telling whether the prover is still present—it only knows that the prover was there at the start of the session. Repeatedly requesting the password after every few minutes of inactivity is hardly acceptable from a usability perspective, so the app must live with a window of vulnerability during which access is granted even though the principal interacting with the app may have changed. How much nicer it would be if the app could sense the presence of the authenticated principal continuously throughout the session, but without imposing an additional burden on the user! The Active Badge [ref] used to do that, locking the workstation's screen as soon as the user left; Corner and Noble [ref] implemented "zero-interaction authentication", based on the computer recognizing the proximity of a token held by the user. The Pico uses short-range radio in order to support "continuous authentication" as a new mode of operation:

– **Benefit 8**: The user is authenticated to the app continuously throughout the session, not just at the beginning. In the course of the session, the app can lock and unlock itself automatically.

Clearly, since this is a new feature, the app requires updating in order to take advantage of it. There is also the issue of relay attacks. Cfr. section 4.3.

3 User authentication with the Pico

3.1 Core design of the Pico

We might choose to bastardize the design later (cfr. section 5) but, at least in its ideal incarnation, the Pico looks and behaves as follows. It is a small, portable, dedicated device, intended to be carried along all the time, just like your key

There, too, the authentication process is started by the verifier and it would be somewhat annoying for the user to have to select that verifier from a list—better to have the device recognize the verifier automatically. However, in this situation, the Pico can actually do much better: see section 4.3 on continuous authentication.

[13] Except if the user has several "accounts" with the same app, in which case further input will be necessary. See section 3.2.

ring. It has a few buttons (possibly "soft" buttons based on a touch screen), a small display, a camera suitable for the acquisition of 2D visual codes and a short-range radio interface. It could be shaped like a smart phone but also like a watch, a key fob, a bracelet or an item of jewellery.

Each app has its own public-private key pair. The Pico talks to the app over radio by sending an encrypted message to the app's public key, thus defeating eavesdroppers. The Pico's message contains an ephemeral public key generated by the Pico, which the app can use to reply over radio[14]. The Pico remembers the public key of the app when it first pairs with it (section 3.2), thus defeating phishers and other men in the middle in subsequent interactions. This is conceptually similar to the Pico doing some kind of SSL with the app, but without a PKI to certify the app's key[15].

This arrangement gives us bidirectional encrypted wireless communication between the Pico and the app, but so far only the app endpoint has been authenticated. To authenticate the Pico to the app, the Pico proves ownership of the credential established during pairing (section 3.2).

The general model of the app is that it requires a *(userid, credential)* pair, which it verifies against its stored validation data in order to authenticate the user. This suits most cases, including some in which the userid is implicit. There are, however, cases that this model does not represent well and that we must treat slightly differently, as discussed in section 3.7.

The app displays a 2D visual code [ref McCune et al, Wong et al] that encodes a hash of the app's self-signed certificate, containing the app's human-readable name[16] and public key. The app has a special "auth screen" (the place where it would normally require a password) that displays that visual code. The user acquires the visual code with the Pico's camera, to identify the app to the Pico, and then, through the Pico, the user invokes one of two actions: either "offer credentials", which is equivalent to typing the password; or "initial pairing", which is equivalent to creating a new account[17].

---

[14] The reason for using an ephemeral key rather than the Pico's long term credential for that app is to protect the user's privacy: the user's identity is only revealed to the app *after* the app's identity has been verified by the user's Pico. When the Pico first talks to the app, by encrypting its message to the public key whose hash is in the chosen visual code, the app has not yet proved possession of the corresponding private key, so it could still be an attacker impersonating the app.

[15] But without PKI, couldn't a malicious app impersonate the genuine one from the first time onwards? Yes, in the sense that the attacker could fool the Pico into believing that her fake app is the genuine one. But she could not exploit that to act as middleperson and open an online banking account at the real bank (app) with the user's credentials, because the visual code authenticator provided by the bank out of band (see end of section 3.2) is tied to the real app's public key. See also the related discussion at the end of section 3.5.

[16] To have something to call it by on the Pico.

[17] I do not expect these to be the actual labels on the buttons of the Pico. We'd have to think a little harder to come up with names that unequivocally make sense to non-geeks.

3.2 Initial pairing between the Pico and a new app

The user acquires the visual code of an app that the Pico has never met before. In that case the only possible action is "initial pairing", which means to create an account[18]. If the visual code belongs to an app for which the Pico already has credentials, both buttons of the Pico are enabled, because in general it is possible to set up several "accounts" with the same app[19].

During the initial pairing phase, if the Pico doesn't already know the app's public key, it requests it and obtains it (in a self-signed certificate) over radio, and verifies that it matches the certificate hash encoded in the acquired visual code[20]; then it stores it in the newly-created memory slot for that account. The Pico generates an ephemeral key pair[21] and sends the public key to the app, challenging the app to prove ownership of the app's secret key.

In the clean-slate design of the Pico, the credential that the Pico holds for each account is a long-term public-private key pair. Once the app proves ownership of the app's secret key, the Pico sends along its long-term public key for that account, which the app returns in a certificate signed with the app's private key. The Pico stores that certificate in the slot for that account. In subsequent "offer credentials" interactions, the Pico will give that certificate back to the app and the app will verify that the Pico owns the matching secret key, for example by making the Pico sign a specially-formatted message containing a nonce supplied by the app.

Arguably, using a key pair as the Pico's credential does not buy us very much compared to just using a shared secret. Using such an alternative, during initial pairing the Pico would first verify that the app knows the secret key matching the advertised public key for the app. Then the Pico would make up a secret bit string and share it with the app, by sending it along encrypted under the app's public key. In subsequent "offer credentials" interactions, the app would verify that the Pico owns the shared secret[22], for example by making the Pico encrypt a specially-formatted message containing a nonce supplied by the app. The usual problems of passwords vs public-key crypto do not seem to affect us here:

[18] Pressing the "offer credentials" button won't work but, interestingly, might trigger a phishing warning, as the user's intention suggests a belief that an account has already been set up with that app. Various heuristics could be used to assess the likelihood of the current app being a phish, including whether its human-readable name is similar to the one of any app registered in the Pico's slot (as in: did the user click "offer credentials" because he wasn't paying attention, or because he thought this was the banking site for which he already has a registration?).

[19] Should this be meaningless for a particular app, a special flag in the visual code could say so, thus disabling the "initial pairing" button.

[20] If not, an attacker might be trying to pass a fake public key for the one of the intended app, so the Pico will abort.

[21] It may have generated several during idle time, to reduce latency.

[22] I try not to call it a password because it is used differently—it is not transferred from Pico to app during "offer credentials".

- The shared secret can't be overheard during authentication because it is not transmitted, and it can't be overheard during pairing because it is encrypted under the app's key.
- The authenticator can't be replayed because challenges and responses are all different.
- A malicious verifier (app) can't use knowledge of the shared secret to impersonate the Pico elsewhere because that shared secret is only used with that app.
- An attacker that somehow managed to break into the Pico[23] would be able to impersonate the user regardless of whether the credential is a shared secret or a private key.

One of the few advantages I see for public key is that, if an attacker breaks into the app and steals the user's credential, she can impersonate the user if the credential was a shared secret but not if it was a public key. However this does not sound like a major concern because she can only impersonate the user with respect to that one app that she has already penetrated; arguably, if she has the ability to compromise the app, she can probably abuse the user's account just by manipulation of the app's private data, without having to resort to user impersonation. Having said all that, we don't appear to gain much by moving to shared secrets either, because the Pico must still support asymmetric cryptography operations to handle the app's public key. A very slight advantage is perhaps that remembering a shared secret is what's needed for backwards compatibility with traditional passwords, hence a design based only on that might be slightly simpler[24].

But back to our regularly-scheduled programme. From the Pico's viewpoint, the account can be seen as a triple *(app, userid, credential)* and therefore, during initial pairing, the Pico and the app must agree not only on the credential but also on the userid. Since the app must already have ways for the user to define a userid, it seems logical to dictate that the userid will be defined on the app side and then communicated to the Pico during pairing over the protected radio channel.

As far as the Pico is concerned, if the app does not prevent it, the user is free to create arbitrarily many accounts with the same app, using different userids. The Pico will create different credentials for each, making these identities unlinkable by the app. (Note how our "purely local authentication" arrangement, which does not use a PKI, protects the user's privacy[25].) During subsequent "offer

[23] Aaargh! We normally assume this can't be done. This may be wishful thinking, though.

[24] Though not by much, since the verification protocols would still be different between the backwards-compatible case and the more secure new one.

[25] If we had asked a certificate authority to sign a hypothetical top-level public key for the Pico (as it would clearly not be scalable for the Pico to ask for a CA certification on each and every one of its hundreds or thousands of app-level public keys as they are created), then the various identities of the user held in the Pico could be linked by the app, or by several colluding apps. This way, instead, even several userids of

credentials" interactions, when the user acquires the visual code of the app, the Pico will notice that it has several accounts (thus userids) for that app and it will ask the user to select one of them. The Pico will send the chosen userid to the app and will then be challenged to prove ownership of the corresponding credential.

Before officially creating the account in its back-end, the app performs whatever checks would have been appropriate in the password world: perhaps none in the case of, say, provision of a new free webmail account; but verification of some other credential that the user obtained out of band if the account connects to pre-existing resources and privileges, as in the case of opening an online banking account to be linked to an existing bank account (where we expect the bank to have already given the user a piece of paper with some authenticator to be typed into the app during the "open a new online banking account" process). Such authenticator must be passed from user to app *via the Pico*, in order to benefit from the protected channel established between them after the SSL-like handshake. There must thus be a way to enter the authenticator in the Pico during the "initial pairing" procedure, and this will be to render the authenticator as a visual code (e.g. in a letter from the bank to the user) and to have the Pico acquire it with the camera. The out of band message must also include the app's visual code, with the semantics that the Pico must send that authenticator only to the app whose public key certificate matches the given visual code, to guard against phishing which is essentially app impersonation.

Once the app is satisfied that the user knows the authenticator and therefore that the credential supplied by the Pico must be granted the privileges of that user, the app records the Pico's credential as valid in its back-end[26].

3.3 Authenticating to a web site app

This is the case that has received the greatest amount of attention, both in the literature and in actual implementations, partly because so many of these new passwords we are asked to remember come from web sites, and partly because the uniform interface offered by the web makes it relatively easy to address (compared to the other cases), for example by augmenting the browser or by using a proxy.

The Pico talks by radio to the app, we said. Actually, the Pico talks by radio to your local computer, which must have an appropriate network stack. From there the messages must go to the browser and then to the app, which is implemented on a remote web site. So for the Pico to work we must also appropriately augment the local computer and the local browser, as well as the

---

the same Pico with the same app will appear to the app as totally independent and unlinkable.

[26] If the Pico's credential is a public key rather than a shared secret, the app may give the Pico a signed certificate containing the userid and the public key and even refrain from storing the credential itself. In any case, as we said above, no PKI is necessary.

app's remote web site itself. We must define clean APIs to allow these radio messages to go between OS and browser and between browser and web page. Having said that, the SSL-like encryption is end-to-end between Pico and app and therefore communications are protected from eavesdroppers and middlemen regardless of the number of intermediate steps.

3.4 Authenticating to a local app on your computer

Examples of this case include logging into a local account on your computer, running an application that requires administrator privileges, switching to another user, unlocking the screen saver and entering the restricted section of the BIOS configuration utility[27].

The generic framework described so far continues to work in this case but the main concern is that the app's secret key must be stored on the local computer. Sensible computer hygiene rules tell us not to leave around secret keys in plain text, because an attacker with physical access to the machine could read them off the raw disk, or even substitute them with her own. But, if we encrypted the app's secret keys, who should provide the keys to decrypt them, and when?

If we supply some authentication credentials to a verifier, we ought to consider the verifier as part of the Trusted Computing Base[28]. If the verifier is not trustworthy it is prudent to assume that the assets that these credentials are meant to protect may be compromised. Indeed, in the pre-Pico arrangement, all the back-end verification data used by the local apps mentioned above *does* usually sit in plaintext on the hard disk, and *is* vulnerable to an attacker with physical access to the drive, so we are no worse off with Pico[29].

We must ensure that a malicious app running with the privileges of my user can't access the back-end authentication data of the other apps, otherwise we could not provide Benefit 5. With OS-level access control mechanisms we make the relevant file inaccessible to normal users and we force the app to go through system calls to access its file. The OS will only grant access after verifying that the request comes from the appropriate app (perhaps based on a hash of the app's binary).

As already mentioned, the Pico talks by radio with the local computer, not directly with the app. So we must provide appropriate middleware (drivers, network stack etc) and APIs, not just modifications to the apps, in order to support the Pico.

---

[27] Note that the last two typically don't require a userid.

[28] "A principal that can violate your security policy", in the cynical but lucid formulation of Bob Morris.

[29] The solution of full disk encryption applies to both cases. If the encryption key is held in a tamper-proof module on the motherboard, the disk is unreadable in other computers—increasing confidentiality but reducing availability, unless special key management precautions are adopted. If the encryption key is provided by the user at boot time, the Pico might, recursively, save some hassle (cfr. section 3.7).

3.5 Authenticating to a non-local but non-web app

Network login and various forms of remote login (including ones involving ssh and Kerberos) fall into this category. So does an email program. Architecturally, the situation is similar to that of web apps. In practice, though, there are at least two important differences.

One, these remote apps are not all accessed through the same local client (as was the web browser in the other case). This suggests that the middleware code through which the browser talks to the Pico should be factored out of the browser and moved further down, in the OS. The browser should access that API to talk to the Pico just like the local client portion of any other non-local app.

Two, some of these apps have a local client that is text-based, without a GUI. This suggests that the visual code generation should also be moved down into the OS. The app supplies its public key and the local OS hashes it and renders it as a visual code.

The case that remains uncovered is that in which the OS isn't running in graphical mode at all and thus cannot render a visual code. In such circumstances the Pico could still acquire the app's public key via radio, but would lack the multi-channel assurance that it's the key of the intended app.

During "initial pairing", this can be remedied by adopting the somewhat elaborate solution we already described in section 3.2 for online banking and in general "accounts that grant access to pre-existing resources and privileges": namely that the back-end will send the user a letter with two visual codes, one encoding the app's public key and another encoding a bootstrapping authenticator. Thus the Pico acquires the app's visual code from the paper letter rather than from the (text-only) screen, and all is well.

During subsequent "offer credentials" interactions, if the local computer is in text mode and cannot display a visual code at all, we will be forced to lower our standards somewhat and blindly accept the public key that comes from the radio without the benefit of multi-channel confirmation[30]. The minor vulnerability opened by this choice is that an attacker could supply another public key and trick the Pico into interacting with that; but this doesn't appear to be a major threat: an active middleperson attack would not work, because[31] the middleperson's public key would be rejected by the Pico as unknown; and supplying the public key of a genuine app with which the Pico had already paired would only cause the Pico to reject the transaction once the middleperson failed to demonstrate ownership of the other app's private key[32].

[30] The "more secure" alternative of requiring the user to select the intended app from the Pico would violate Benefit 7.

[31] Aside from the physical difficulty for the attacker in keeping the radio connections to the Pico and the real app separate.

[32] If the attacker were the owner of the other app with which the Pico had already paired (a kind of insider fraud) then OK, the protocol would go through and the Pico would authenticate to the wrong app; but the genuine app would never receive the expected credentials and would refuse access to the corresponding resources,

3.6 Authenticating to a non-computer app

Authenticating to a non-computer app covers cases such as having to type an unlocking PIN into a car stereo, a burglar alarm, a cash dispenser, an electronic safe and so forth.

Under the unrealistic clean-slate assumption that we are allowed to equip the non-computer device with the appropriate radio interface and to update its firmware, the situation is similar to the one, just seen in section 3.5, of computer-based apps without a graphical display. The only minor difference, which is of course trivial to handle, is that here there would be no userid.

3.7 Uses that aren't really authentication

Consider the case of a program that performs symmetric encryption and decryption on files. For sure it requests you to remember and type a password (potentially a different one for every file) and as such it must be catered for by the Pico, otherwise we'd be failing to offer Benefit 6.

But try to model it as we have done so far and something doesn't feel quite right: would every file be a different app? Or would the encryption program be the app, and every file a different userid? What if I wanted to encrypt the same file under five different keys, to give it to five different recipients—should the userid be my own annotation, such as "file *X* that I'm sending to recipient *Y*"? More importantly, if my Pico proved ownership of the correct credential for the chosen file, how would the app unlock the file for me? Would it authorize use of a decryption key stored next to the verification data for my credential? Surely that's broken: an attacker with physical access could read out the decryption key in plaintext, defeating the point of storing the file in encrypted form.

There is indeed a crucial conceptual difference between this case and the previous ones: the file encryption application, even though it asks for a password like all the others, is not *authenticating* the user, nor is it *verifying* any credentials whatsoever: it is just *accepting* a bit string from the user and deriving from it a cryptographic key with which to decrypt the selected file. This app, unlike the others, wouldn't even *know* how to unlock the user's assets (i.e. the encrypted file) without the user-supplied string—it's not the case that the app has access to the data but won't grant it to others until they prove they're worth it. And it's crucial that the app *not* be able to decrypt—the information necessary from decryption must be supplied by the user, not remembered by the app. It's a subtle but significant distinction, especially if you think about the one-time pad where different decryption keys yield different plaintexts for the same ciphertext, all equally valid a priori.

For this kind of application, the proper thing for the Pico to do is to disregard the whole business of credentials (and public-private keys for the Pico) and

which the user would notice. It is thus unclear what advantage the attacker would gain other than a mild denial of service, not much different from what it could in any case do by jamming the radio channel.

literally just to remember a "password" and spit it out on request. We will still benefit from the SSL-like arrangement in order to prevent eavesdropping and we'll have to make sure that the password transfer protocol involves some randomness, to prevent replay attacks.

Sadly, it seems to me that we'll probably have to give up on Benefit 7 here.

4 Details of Pico operation

4.1 Locking and unlocking the Pico

It would clearly be unacceptable to allow anyone in physical possession of the Pico to use the credentials it contained. We want the Pico to be locked when not in possession of its owner. Moreover, while the Pico is locked, we want its secrets to be unrecoverable even by adversaries who can disassemble it and inspect its memory bit by bit.

In my clean-slate design, the non-volatile storage of the Pico is fully encrypted, with a key that the Pico forgets when powered down. I would be violating Benefits 1, 4 and 6 if I asked the user to remember that key, so the key to unlock the Pico is instead recovered by n-out-of-n secret sharing, with shares held by other small radio-enabled devices usually worn by the user such as special versions of glasses, watch, belt, pen, cellphone, wallet, keyring and so forth—the Picosisters.

The Pico periodically polls the Picosisters in range for their key share, using a protocol[33] that ensures that they only answer after recognizing the Pico, and that eavesdropped answers can't be replayed and don't reveal the key share.

When the Pico hears a share, it remembers it; but it activates a decay (countdown) timer for that share that tells the Pico when to forget it (e.g. after a minute or so). When a share is heard again, its decay timer is topped up to the starting value. When the decay timer expires, that share is erased. As soon as the Pico has fewer than n shares, it forgets the shared secret and stops being able to authenticate[34]. Unless it reacquires the missing shares within a set timeout, the Pico switches itself off, thus forgetting *all* shares, and must be explicitly turned on and unlocked before it will work again. (The user may of course switch off the Pico intentionally even while all the Picosisters are in range.)

Two of the shares are special and have a much longer timeout (e.g. a day). One of the special shares it is not obtained from a Picosister but from a biometric[35], with suitable error correction [ref Feng]. The other special share is obtained

[33] To be invented. Also to be invented is the initial pairing between Pico and Picosisters, together with the share generation procedure.

[34] This also applies to continuous authentication (section 4.3).

[35] The biometric as an additional authentication factor has the advantage of usability (and few of the privacy concerns normally associated with biometric authentication, because the verifier is your own device) but won't be as strong as the published statistics on biometric authentication reliability might suggest because here the verification is not supervised by a human verifier suspicious of the prover and thus it is not resistant to an adversary who has control of the Pico and feeds it iris photographs

from a remote server (conceptually belonging to the user) through a network connection.

Turning on the Pico requires the presence of all n shares: the $n-2$ Picosisters, the biometric and the remote server must all contribute. The idea is that, even if all the tokens (Pico and Picosisters) were stolen together, for example while the owner is swimming on the beach, it would be hard to reactivate a locked Pico.

The special share held by the remote server has a dual purpose: auditing the reactivations (the server keeps a log of all the times it sent out its share, and to which network address) and allowing remote disabling of the Pico (the user who loses control of the Pico can tell the server not to send out the share any more). I really like this idea! :-)

4.2 Coercion resistance

If your threat model is that you may be kidnapped by the mafia or the secret police and tortured in a dark basement until you reveal your passwords, you'll be happy that in my design the Pico isn't unlocked by a password. Having removed the dependency on "something you know", there is no incentive to torture you[36].

If you succeed in destroying the Pico (or even just one of your Picosisters) while the bad guys are capturing you, all the credentials that your Pico protected are safe[37].

If you are James Bond on a special mission and suspect that you might be captured[38], you may wish to trade off some availability in exchange for security and disable the remote server, after setting the timeout of the remote share to the duration of your trip. You won't be able to reactivate your Pico until you get back home if it ever goes off, but neither will your captors. Now all you need to do while being captured is to switch off the Pico, not even to destroy it.

Another thing you can do if you need to protect some very precious credentials (and perhaps if you need to be able to deny that they even exist) is to dedicate *a separate Pico* just to them. You'd have your ordinary Pico for all the damn passwords you have to use on a day-to-day basis, and then this other special Pico that nobody would know about, kept in a safe place and less likely to be misplaced or stolen, for the more serious and rarely accessed stuff.

---

or gummi fingers [ref] without the Pico knowing that they are forgeries. The biometric verification process should still make at least a crude attempt at verifying that the biometric is live.

[36] Of course they might still torture you just because they want to anyway—there is no guarantee that they will be rational about it. But at least we are removing the main motive.

[37] And you will later be able to recover them using your backups (section 4.4), which of course you keep "in a safe place".

[38] Or, more mundanely, if you are on a trip away from home and think there's a higher than usual risk that you might lose your Pico.

4.3 Continuous authentication

As previously announced, a major advantage enabled by the Pico is *continuous authentication*: once you have "logged in" by acquiring the app's visual code and letting the Pico do its stuff over the radio, the app can continue to ping the Pico over their confidentiality- and integrity-protected short-range radio channel and use that to confirm that you are still around. As soon as you are not, the app can block access[39]; and, what's even better, if your Pico reappears before a set timeout expires, the app can grant access again without even asking you to reacquire the visual code.

One potential threat in this situation is the relay attack, which an adversary could use to make it look as if you were still close to the computer even after you left. Even though it is difficult to construct a plausible scenario for this attack, especially given that the "offer credentials" phase involves a multi-channel protocol, it will still be wise to protect this continuous authentication with a suitably robust distance-bounding protocol [ref so near and yet so far].

Engineering details to be addressed include state preservation (things must work like an automatic suspend/resume where you still find everything just as you left it, not like an automatic logout/login where all open programs get closed) and the nesting of app sessions (one of the apps may be your actual login session on the local computer and another might be your webmail session within your web browser; each with its own key pairs and so on; we must be careful about the interaction between those sessions if the suspends and resumes don't properly nest; moreover, the same computer supports many apps but just one physical radio interface, so we must clarify the low level protocols of which app gets to offer their certificate over the physical radio channel, and when).

4.4 Backup

To protect the user from permanent loss of access if the Pico is destroyed or lost, correct operation of the Pico requires regular backup.

At home (or anywhere else that you deem "a safe place") you have a docking station, to which the Pico is paired[40], that takes a backup[41] every time you dock the Pico, which you must do regularly to recharge the battery anyway (as with your phone).

There is a procedure for restoring the encrypted backup onto a virgin Pico, after associating the virgin Pico to the docking station and to the Picosisters. Of course you can still only access this backup by unlocking the Pico in the usual way as per section 4.1.

---

[39] Like a locked screen saver that came on *only* when you left, not merely when you stopped typing for a while; and *as soon* as you left, not merely half an hour later.

[40] Details to be defined, but possibly via Resurrecting Duckling [ref].

[41] As we said in section 4.1, the mass storage of the Pico is encrypted. For backup you therefore just dump the encrypted content of the Pico's mass storage as is.

The docking station has all the appropriate interfaces (radio, network, USB, memory card, whatever) to allow extraction and external storage of this encrypted backup.

There is also a procedure for splitting the key of an unlocked Pico into a new set of n shares, and for backing up these shares through the docking station for external storage (hopefully not all together!).

Some good usability ideas are needed here to ensure that backups will be taken regularly[42] and that the saved shares will be stored in safe and independent places.

5 Optimizations (as Roger Needham would call them)

"Optimization", Roger Needham famously remarked, "is the process of taking something that works and replacing it with something that almost works, but costs less". There are various ways in which we could "optimize" the design of the Pico in an attempt to bootstrap its acceptance and market penetration.

5.1 Using a smart phone as the Pico

One of the "costs" of the Pico is that non-geeks won't like to have to carry yet another gadget. To counter that, the Pico can be implemented as a smartphone program: the smartphone has a good display, a visual-code-compatible camera, a radio interface and, above all, is a device that users already carry spontaneously. The main trouble with this option is that the smartphone is a general-purpose networked device and thus a good environment for viruses, worms, trojans and other malware. Personally, I would not feel comfortable holding all my password replacements there. It also remains to be seen whether an unprivileged smartphone program is granted enough low level control of the machine by the OS to be able to encrypt all of its data without inadvertently leaving parts of it in places that an attacker could re-read.

There is however some merit in using a smartphone as the Pico, given the user acceptance of the smartphone device and the ready availability of the hardware. The more paranoid users who would have preferred a dedicated device could even use their *old* smartphone as their Pico (wiping it of any other apps and restricting network functionality to getting the special share from the remote server as described in section 4.1) and their new one as their regular phone.

5.2 Typing passwords

The most significant "cost" of the Pico is undoubtedly the fact that it requires changes to the apps. Next to that, the fact that it requires hardware and software radio support on the local computer (as well as on the Pico itself). A way to

[42] The "dock to recharge" idea is a good start but may not be enough—what if I need to recharge when I'm not in a "safe place"? What if I use spare batteries?

eliminate all these costs is to forget the public key and visual code for the app and all the SSL-like business, and to make the Pico just remember straightforward passwords. With a USB connector[43] the Pico could emulate a keyboard and type the password in your stead, honouring Benefit 4.

The ways in which this optimization works only "almost" are unfortunately many. Giving up the SSL-like operation opens the door to app impersonation, losing Benefit 5. Because the app no longer has a visual code or public key, the Pico can't select the appropriate credentials, losing Benefit 7. Because the app can no longer ping the Pico, we lose Benefit 8. All we can retain, provided that we manage the passwords sensibly, are Benefits 1, 2, 3, 4 and 6.

Besides that, passwords typed by a simulated keyboard could be captured by a software keylogger if the local computer were affected by malware[44].

An even more radical cost saving measure would also do away with the USB cable, forcing the user to transcribe the passwords manually and thus giving up even on Benefit 4. This would essentially bring us back to the PDA with password wallet software, offering only Benefits 1, 2, 3 and 6.

5.3 Local cost cuttings

Further savings could be obtained by dispensing with some of the administrative features: by not bothering with the Picosisters (section 4.1) or simply replacing them with a PIN (horror!); by dropping proper mass storage encryption (section 4.1); or by not implementing the automatic backup from the docking station (section 4.4). I regard these as false savings because they only affect the client end and thus they don't help us gain Pico support from more apps, whereas they substantially reduce the security and availability of the Pico.

6 Related work

In the 1990s, crypto-geeks would use the crude solution of storing their passwords in a text file encrypted with PGP and preferably held on a non-networked computer. It offered only Benefits 1 and 6.[45]

One of the earliest purpose-built "password wallet" applications (circa 1999) was Schneier's Password Safe [ref]. It would let you organize passwords by categories, protect them with a master password and allow cut-and-paste into Windows programs with standard clipboard commands. This offered Benefits 1, 4, 6.

In the early 2000s, crypto-geeks would use similar programs but running on a PDA. The PDA was more portable than a laptop, and more secure owing to

[43] Or a clever device driver—but that requires installing software on the local computer.

[44] Though that's no worse than what could happen today to human-typed passwords, vulnerable not only to software keyloggers but even to hardware ones, hidden in the keyboard or keyboard connector, that software can't even detect.

[45] Here and elsewhere, one might argue that Benefits 1 and 6 are only "almost" provided, on the basis that the user is still required to remember one master password.

the lack of a network connection, but less convenient because it did not allow cut and paste. This went back to providing only Benefits 1 and 6 (plus, if we are generous, 2 and 3, to the extent that the program could, on request, generate a random password).

In 2002 Corner and Noble [ref] described their design and implementation of "Zero-Interaction Authentication", a system in which a laptop with a cryptographic file system was augmented with a wireless authentication token worn by the user, providing a high level of protection without having to type any passwords[46]. Proximity of the token would unlock the *keys* used to decrypt the files on the laptop. Absence of the token would flush the decrypted files from the cache and leave their decryption keys inaccessible. This system was not meant as a general-purpose password replacement but it provides a well-engineered example of how to offer Benefit 8. Within the limited realm of replacing just one specific laptop-unlocking password, it also offers Benefits 1, 2, 4 and 5.

A brilliant solution for handling web passwords, the Mozilla Firefox Password Manager [ref], was introduced in the mid-2000s. This feature, integrated with the browser, collected (with your permission) your userid and password as you typed it into a web page and later pre-filled those fields on your behalf whenever you visited that page again. This added Benefit 7 on top of Benefits 1 and 4 (a significant progress for usability), even though it obviously dropped Benefit 6 because the system only worked with web pages. There had been other instances of browsers remembering passwords, but this one allowed you to protect the stored passwords with a master password (XXX was it the first to do that?). The system occasionally fails to recognize that a page is requesting a password, and occasionally produces unexpected results during password update; but on the whole it is commendable for both usability and security[47].

OpenID, introduced in 2005 [ref], is a web standard whose main purpose is to allow one user to log in to many web sites using a single identity and credential, but without having to share the same password with all sites. It is a different, cloud-based approach to "reducing all your web passwords to one" (as the Firefox Password Manager effectively does, but with local storage) and as such provides a similar set of Benefits: 1, 2[48], 3 (main purpose), 4, 7. But it has the significant privacy disadvantage that your OpenID provider gets to know about every OpenID-based web site you visit. Similar comments apply to the proprietary but more widely deployed Facebook Connect.

In 2006 Parno, Cuo and Perrig [ref] described their design and implementation of "Phoolproof Phishing Protection". Their objective is to guard against phishing: elimination of passwords is only a beneficial side effect. Their system,

[46] Though note that the token itself had to be activated once a day with a PIN.

[47] For a security person, judging from personal experience as help desk for friends and family, it is depressing to note how few ordinary people are brave enough to use it, even after encouragement, despite it being so well designed and so easily accessible.

[48] Debatable. The system prevents the user from picking weak passwords for all the other web sites, but—here and elsewhere—nothing stops the user from choosing a weak master password.

intended to protect online banking website accounts, introduces many features that the Pico also adopts: it involves a cellphone interacting via Bluetooth with the local computer, a web browser plugin to talk to the radio and a full SSL interaction, with mutual authentication, with the remote web server. It offers Benefits 1, 2, 3, 4, 5. They do not offer Benefit 7 because of an explicit security-inspired design choice that the user must select the app from a list on the cellphone[49], with the cellphone then instructing the browser to visit the appropriate URL.

7 A possible roadmap for future work

Assume adequate funding and manpower are available. How to get rid of passwords using Pico? What's the grand plan for world ~~domination~~ liberation?

The first task will be to define precisely all the sub-protocols (Pico pairing with app, Pico offering credentials to app, Pico collecting shares from Picosisters, biometric and remote server, Pico interacting with docking station for backup etc etc) and to implement a working prototype of all the components (Pico, app, middleware on local computer, Picosisters etc). Initially, all components will be implemented on general-purpose computers: the point at that stage will not be to evaluate usability but to discover subtleties and edge cases[50] that may have been missed in the current broad-brush design. This system will be self-contained and will not work with existing apps. Implementation will be followed by an adversarial security review.

The next phase will be to prototype some practical cases. The Pico will be implemented on a smartphone and will also support existing unmodified apps in a gracefully degraded way, by typing the password as a simulated USB keyboard or, at worst, by letting the user transcribe it. On the app side, a few open source apps will be converted, both as feature demonstrators and as programming examples, covering at least one app in each of the computer-based categories mentioned in section 3 (e.g. a blogging platform; a local login program; an email client; and a file encryption program). It will be OK to have clumsy-looking Picosisters and docking station and to forgo the biometric. All the features described in section 4 (continuous authentication, locking and unlocking with secret sharing etc) will be supported. Implementation will be followed by user studies, performance analysis and, potentially, redesign.

The redesigned system will then be re-engineered, producing reference software libraries (initially only for one OS) to make it easy for third parties to support Pico in their apps. Most importantly, an RFC-like specification will be produced, precisely describing a compliant implementation of each component.

Realistically, the Pico will probably remain a smartphone program, rather than a dedicated device, until enough real-world apps support it; so the next major effort will have to be to build critical mass, both for users (by making it a compelling free install on as many smartphones as possible—something useful

[49] Foregoing Benefit 7 means they don't need a visual code, which makes their protocol more compatible than the Pico's with existing deployed infrastructure.

[50] Including important "exceptional events" such as credential revocation.

and relatively secure even just as an old-style password wallet) and for apps (by making it easy and secure to integrate our reference authentication module in their code and by winning over the providers of a few leading apps that people use every day—webmail, computer login etc).

At that stage we will finally reconsider whether the smartphone implementation of the Pico is satisfactory or whether we couldn't do better, in terms of usability and security, by porting our software to custom hardware, despite users' aversion for carrying yet another gadget. If yes, we will provide a reference design that manufacturers will be able to adopt, improve and commercialize.

The medium-term goal is to reach a stage in which most apps that used to request a password now also support the proper Pico-based replacement side by side. We don't mind remaining in that stage for a long time, because then every user who wants to reap the security *and* usability benefits of the Pico can do so regardless of whether others also do. Eventually, in the long term, once app providers observe that most of their users find it more convenient to authenticate with Pico rather than passwords, they will be able to retire passwords for good.

8 Conclusions

It is no longer reasonable for computer security people to ask users to remember passwords, because there are too many and because the passwords that would be strong enough are too complicated for human memory.

Pico is perhaps the first proposal of a user authentication solution that would eliminate passwords altogether, not just for a special category of apps.

Overcoming the backwards compatibility shackles will be very hard, but non-geeks (and geeks too) deserve a more usable and more secure solution than what we're offering them now.

Acknowledgements

I am indebted and grateful to Virgil Gligor and Adrian Perrig for their generous and intellectually stimulating hospitality, as the first draft of this paper originated while I was on sabbatical at CMU Cylab in 2010.

References

Those in the text plus at least

Cormac Herley, Paul C. Oorschot, and Andrew S. Patrick. Passwords: If We're So Smart, Why Are We Still Using Them?

Dinei Florêncio, Cormac Herley, and Baris Coskun. "Do Strong Web Passwords Accomplish Anything?" `http://www.usenix.org/event/hotsec07/tech/full_papers/florencio/florencio.pdf`

Gilbert Notoatmodjo and Clark Thomborson. Passwords and Perceptions.