Pico without public keys

Frank Stajano[1], Bruce Christianson[2], Mark Lomas[3]
Graeme Jenkinson[1], Jeunese Payne[1], Max Spencer[1], Quentin Stafford-Fraser[1]

[1] University of Cambridge
[2] University of Hertfordshire
[3] Capgemini

Abstract. Pico is a user authentication system that does not require remembering secrets. It is based on a personal handheld token that holds the user's credentials and that is unlocked by a "personal aura" generated by digital accessories worn by the owner. The token, acting as prover, engages in a public-key-based authentication protocol with the verifier. What would happen to Pico if success of the mythical quantum computer meant secure public key primitives were no longer available, or if for other reasons such as energy consumption we preferred not to deploy them? More generally, what would happen under those circumstances to user authentication on the web, which relies heavily on public key cryptography through HTTPS/TLS?

Although the symmetric-key-vs-public-key debate dates back to the 1990s, we note that the problematic aspects of public key deployment that were identified back then are still ubiquitous today. In particular, although public key cryptography is widely deployed on the web, revocation still doesn't work.

We discuss ways of providing desirable properties of public-key-based user authentication systems using symmetric-key primitives and tamper-evident tokens. In particular, we present a protocol through which a compromise of the user credentials file at one website does not require users to change their credentials at that website or any other.

We also note that the current prototype of Pico, when working in compatibility mode through the Pico Lens (i.e. with websites that are unaware of the Pico protocols), doesn't actually use public key cryptography, other than that implicit in TLS. With minor tweaks we adopt this as the native mode for Pico, dropping public key cryptography and achieving much greater deployability without any noteworthy loss in security.

1 Introduction: a motivating story

In 2013, an Adobe authentication server was famously broken into [1]. Consequently, every one of the 150+ million customers whose credentials were leaked was forced to change their password. Why? There are several reasons, some resulting from Adobe's carelessness and others that are more fundamental.

Authors' preprint of 2015-06-30.
Final version to appear in *Proceedings of Passwords 2014*, Springer LNCS.

Adobe's sins in this affair are numerous and have been widely publicized. To begin with, although the passwords were encrypted[4], password hints were stored in plaintext next to each password. The fields stored in the database for each password included the following:

- User ID
- Username
- Email
- Password *(the only encrypted field)*
- Password hint

If I chose an especially stupid hint (such as "try qwerty123", which actually occurs in the leaked data), my password would now be known to the attackers. But, with an easy password (such as "12345678", which also occurs many times in the leaked data), even if my own hint wasn't totally stupid I would probably be in trouble: if my password was insufficiently original, so that several other people among those 150 millions of victims also happened to choose it, then chances are that *someone else* provided a stupid hint for it. If so, because the passwords were encrypted with ECB and without salt, the attacker could read off the stupid hint from one of them and apply it to all the other passwords that encrypted to the same ciphertext. Moreover, even if no hint obviously exposed the group of people who shared my easy password, if thousands of others share my "12345678" stroke of genius, chances are it's one of the handful of popular passwords[5] that are well known from many other leaks and that attackers try before all others.

So let's stop flogging the Adobe dead horse. Assume that some imaginary site, say `bettersite.com`, had hashed and salted its passwords, as should be best practice ever since Morris and Thompson's 1979 landmark paper [2], and that they imposed a password policy requesting uppercase, lowercase, digits and so forth in order to enforce diversity and prevent those easily guessed common passwords. Then, first of all the policy does not even let you use a really terrible password such as "12345678"; second, even if you do pick an easily guessed password that is chosen by thousands of other clueless users, thanks to the salt it will appear in the credentials database as thousands of *different* salted hashes, so that the attackers can't tell that thousands of people share that same password, nor which ones of them do. It is true, however, that the attackers can still try the most likely guesses that fit the policy. Chances are they'll find that user John chose his dog's name and birth year: "Bella2008" complies with the policy but can be guessed relatively easily in an offline search. So, when their credentials file is compromised, the operators of `bettersite.com` still have to ask each of their customers to change their password. This is not merely because the attacker

[4] Reversibly encrypted with 3DES, not salted and hashed as they should have been. To add the insult to the injury, the encryption was performed by treating each block independently, in ECB mode.

[5] Splashdata publish an annual list: the most recent one at the time of writing is `http://splashdata.com/press/worst-passwords-of-2014.htm`.

could otherwise log into John's account at `bettersite.com`, but also because, knowing many people's propensity for reusing the same memorable password on several sites, the attacker will now try "Bella2008" at John's accounts on Amazon, Ebay, Google and so forth. So John had better change that password on all the other sites where he used it[6].

But perhaps the most annoyed user of `bettersite.com` would be Emily the Geek, who diligently took all the password recommendations very seriously and went to the trouble of composing and memorizing a strong password along the lines of "caeF@#qSFQH?T!@$YF". Since `bettersite.com` salted and hashed it, she is quite safe from guessing attacks, even if the bad guys have a basement full of graphics cards to devote to their cracking. And yet, when `bettersite.com` is hacked, the site operators will still ask Emily to change her password, because they can't tell that she had a strong one and shouldn't have to change it[7]. That's really unfair towards her, after she went to so much trouble, and more so because we know it's quite unnecessary.

Would life not be so much better if websites accepted public keys instead of passwords? Neither John nor Emily would have to change their credential at the compromised website nor at any other, even if they gave their same public key to every site. By stealing a file of public keys, the attackers would not be in a position to impersonate any of the users anywhere—neither at the attacked site nor, a fortiori, elsewhere. This alone looks like a strong incentive for websites to accept public keys instead of passwords.

So let's imagine moving to public keys instead of passwords for user authentication. That was Pico's original plan [3]. With that strategy, stealing the credentials file does not in itself compromise any accounts. The website does not have to force any password (or private key) resets in the event of a breach and, perhaps more importantly, doesn't make front page news of the Financial Times with the consequent drop in share price.

There is one additional benefit of adopting public keys instead of passwords for user authentication: assuming (as is common practice with, for example, PGP) that Emily encrypted her private key with a passphrase, then even an attacker who got her public key by hacking into `bettersite.com` would not be

[6] This may not be part of the advice John receives from `bettersite.com` when they ask him to change his password there, since they have no incentive about protecting his other accounts, despite the fact that it is the leak at their site that has made John vulnerable to fraud elsewhere. But they might argue that it was John's reuse of his password that put him at risk and that, since he contravened their advice against it, it was his fault. We do not condone this buck-passing, since demanding that users remember complex and distinct passwords for each site is unreasonable in the first place.

[7] A better designed password mechanism should blacklist all passwords that have appeared in a document, at least as far as the maintainers of the system are able to figure out. If they knew about this paper, for example, then even Emily's password would no longer be acceptable. VAX/VMS, developed in the 1970s, rejected all of the password examples that appeared in its own documentation.

in a position to verify any guesses of her passphrase (unless he also obtained her encrypted private key from her computer). But this is an aside.

Another aside is that Pico's unconventional strategy of using a *different* public key for each website that Emily visits, rather than the same one for all, gives Emily an extra privacy advantage: even colluding websites cannot correlate her visits[8].

Someone might however object that these benefits (no accounts compromised if credentials file stolen, etc etc) don't really derive from public key cryptography but from the fact that we're now using unguessable credentials. The core of this objection[9] is that, if one could assume the availability of the same mechanisms on which Pico relies, namely tamper-evident tokens to hold the principals' secrets, then one could offer the same benefits without using public keys at all.

This paper explores this alternative and its accompanying trade-offs.

2 Objective

We like the crucial property, which Pico has, that a compromise of the credentials file at Adobe does not expose the accounts at Facebook; or, more strongly, that the compromise of one website does not expose any credentials that could be used for future login, whether at that same website or at any others. We wish to offer this property without using public key cryptography. Why? Perhaps because the mythical quantum computer might one day break public key cryptography. Or perhaps, less dramatically, because in mobile and wearable computing we want to conserve battery energy while performing user authentication. Or maybe for yet another practical reason related to deployability, as we shall see in section 5.

We assume that users (and websites) have tamper-evident tokens that can securely remember as many high-entropy secrets as needed.

3 The core idea

The basic idea is very simple. Assume the website has the decency of salting and hashing the passwords properly[10]. For each website, the user's token picks a different, strong and totally random password of generous length[11] and remembers it. The user never has to type it or even see it[12]. The strong password

[8] This corresponds to having different passwords for the different site, but without Emily having to remember them because the Pico does that for her.

[9] Which was the germ of the discussion among the first three authors that eventually resulted in this paper.

[10] Although we know that this often doesn't happen, as documented by Bonneau and Preibusch [4].

[11] Unfortunately inconsistency between password policies may frustrate this. Some websites reject strong passwords on spurious grounds such as suggesting that they are too long. Our PMF specification [5] addresses this issue, as briefly summarized in "Level 2" in section 5.1.

[12] Some people, but not us, no longer call this a password, because users can't remember it. Note that, if the attacker can perform any industrial-sized number of guessing

is only stored in salted and hashed form, so it can't feasibly be guessed: even though the attacker could verify a correct guess, the search space is too large. The compromise of the credentials file therefore does not expose any passwords and does not affect the attacked site nor any other sites. If "legacy users" (with human-remembered and thus guessable passwords) coexist with "secure users" (with strong random passwords that are too long and complex to remember in a brain, but which a tamper-evident token remembers for them), then, so long as the website can tell which is which, in case of compromise only the legacy users need be required to reset their password.

In fact we have already taken this route with Pico when we implemented the compatibility mode we presented at this workshop last year [6], which essentially transforms Pico into a password manager in your pocket. Since then, we have enhanced our system with the "Password Manager Friendly" (PMF) specification for websites [5]. The changes at the website side to achieve PMF compliance are minimal[13] but they allow Pico (as well as any other password manager, whether stand-alone or embedded in a browser) to reliably create and submit a strong uncrackable password for the website without user intervention, and of course to adopt a different one for every website. PMF almost[14] removes the need for the public-key protocol (Sigma-I [7]) originally used by Pico, so long as the Pico has a secure channel[15] over which to send the password to the website.

3.1 A small leftover problem

The last point we just mentioned needs further attention. How can the user, or the user's tamper-evident token, send the strong password to the website?

attempts offline, most passwords that the average user will actually remember are vulnerable.

[13] They consist essentially of a set of semantic labels for the HTML of the login form, saying in a machine-readable format that this is a login form, this field is the username, this field is the password and so on. The HTML5 specification contains something similar and some web browsers recognise this. For example, you may specify a policy that when web forms are cached any password fields are omitted from the data that is cached.

[14] Why "almost"? Because, with public key, there is the additional benefit that the website cannot impersonate the user. This does not affect accounts at other sites if the user adopts a different password for each, but it does affect logins at the site itself. Without public key, the website, which on exit from the TLS tunnel receives the user's password in plaintext even though it is not supposed to store it other than salted and hashed, could pretend to a third party that the user authenticated at other times. With public key, it would not be able to produce evidence of any other logins than the ones in which the user actually replied correctly to the challenge. As with other arguments in favour of public key, this one too becomes problematic once the possibility of revocation is accounted for. It should also be noted that a better web login scheme would allow the prover to demonstrate knowledge of the shared secret without revealing it—cfr "Level 3" in section 5.1.

[15] In this context, secure means offering confidentiality, freshness and authenticity of source and destination.

In Pico, even in compatibility mode, that's done using the TLS channel that protects the HTTPS login page. But that itself uses public key cryptography. If we want a solution that completely avoids public key, we must use an alternative.

This is not a problem specific to Pico: the scope is much broader. If we assume we can't use public key crypto, how does the web work? How do we do HTTPS? How can a user communicate securely with (let alone log into) a website that they have never seen before[16]?

The rest of the paper will pursue two threads. First (section 4) we revisit the problem of web login without public key technology, which will become topical again if quantum computers become capable of factoring numbers a little larger than 3*5. Next (section 5) we sketch an alternative architecture for Pico that uses as little public key technology as possible, pointing out its advantages and disadvantages compared to the previous design.

4 Web login without public keys

The technology for establishing secure connections with unknown principals without using public key cryptography is well-known (Needham-Schroeder [8], then upgraded to Kerberos [9]). Whereas, with public key, the client can contact the website directly and check the validity of the site's public key certificate, with symmetric key you need an introducer (the Key Distribution Centre, or KDC) with whom both parties share (separate) secrets.

The symmetric key faction will be quick to point out that this requirement is no worse than with public key, because there you need an introducer too: the Certification Authority that signed the website's public key.

The opposite faction will now object that, without public key, the introducer (Kerberos authentication server) must be online, so public key is better because it does not require that of its introducer (the Certification Authority).

The symmetric key faction will then ask how their opponents deal with the case in which a private key has been compromised and a public key must therefore be revoked. If you need to check for revocation online at every login, that's no better than having to talk to the Kerberos KDC at every login. And, if you don't do that, you only have the *illusion* of superiority while instead you're vulnerable for a time window whose length depends on the expiration frequency of the issued certificates.

These issues and trade-offs were extensively discussed in the 1990s, mostly as debates at crypto conferences, but remarkably little of this discussion seems to have been published for posterity in the open literature, if we except a discus-

[16] Note that we are not even beginning to address the even more complex human factors problem that the average user can't understand the difference between the HTTPS padlock in the address bar (whose appearance changes between browsers and between versions of the same browser anyway) and a bitmap of a padlock in the client area of the page, and can therefore be phished.

sion by Christianson, Crispo and Malcolm [10] in a previous Security Protocols Workshop[17].

So let's have a closer look at how revocation is handled on the web today, twenty years later.

4.1 Revocation on the web today

Once credentials are suspected (or known) to have been compromised, they must be revoked and replaced with new ones. This tends to be a weak point in many systems. Even deployed ones. Even on the web as we know it today.

In the original Pico system [3], each Pico token follows the unconventional strategy of using a separate key pair for each website it talks to, which makes it relatively easy for the Pico to know whom to contact if revocation is needed. On the TLS-based web, instead, in which a website offers a public key to all its correspondents (the traditional way of using a public key), it is this widespread sharing of the public key that makes revocation hard. When a principal uses the same public key for all correspondents, and this public key is distributed promiscuously[18], then, when this public key is revoked, the principal can never be sure of having warned all correspondents that the previously used public key should no longer be used. This is particularly true for *future* correspondents, who might begin to use a previously known-good public key without awareness that it has meanwhile been revoked.

In today's web, it's usually only the website that offers a public key to the client, and not vice versa, because client certificates are not commonly used. As we noted, the website's public key is hard to revoke because many correspondents use it, whereas the Pico uses a different credential for every website (regardless of whether it is a public key, in native mode, or a strong password, in compatibility mode) and that makes credentials relatively easy to revoke. If you lose (or discover evidence of tampering on) the tamper-evident device, rather than the individual credential, then you need to revoke *all* the keys (or pass-

[17] It was a common discussion theme at conferences such as Oakland in the mid-1990s, and the subject was also discussed by the Internet Research Task Force (IRTF) Privacy Task Force in the late 1980's/early 1990's. It was widely understood that both symmetric key set-up cost and public key revocation cost were of order $k \log N$ with N counterparties in the case of hierarchical servers. Possibly for economic reasons the early CAs pushed for short certificate life, rather than effective revocation mechanisms, contributing to the rather unsatisfactory situation that we have today. Interestingly, version 1 of the SWIFT protocol used a trusted logger in conjunction with symmetric keys to prevent message forgery, and 3GPP has a current Generic Authentication Architecture that is based on symmetric-key. The authors would be delighted to hear from any readers who are aware of other publications that address in a measured way the trade-offs between symmetric and public key in the presence of revocation.

[18] For example via public key certificates propagating in an uncontrolled way.

words) it contained and that's a pain[19]. But at least you have the exact list of the correspondents you should contact. The website doesn't even have that luxury, because the same public key is used by all clients—including the future clients that the website doesn't even know about.

In today's web, for the website, revocation essentially doesn't work. What happens? A site gets hacked, as happened to Adobe. It had a public key certificate valid until next year, but it makes itself a new key pair because the hackers, who exfiltrated the passwords file, might have done the same with the private key. If the website does not revoke its public key then the attackers, who have grabbed the old key pair and have a still valid CA-signed certificate for the public key, can impersonate the website to any client whose connection they can man-in-the-middle[20].

There are essentially two alternatives to prevent this attack, essentially corresponding to client pull and server push respectively.

The first alternative is for the client to check every certificate with the CA to see if this is still valid, as per the Online Certificate Status Protocol (OCSP) [11]. This adds latency and introduces brittleness: what should the client do if it can't contact the CA, either because of non-malicious network errors or because of a purposeful denial of service? If the client is paranoid and rejects the connection to Adobe whenever it can't contact the CA, essentially the web stops working, as per Leslie Lamport's old adage that "a distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable"; if instead the client accepts the connection anyway, then impersonation attacks go through[21].

The second alternative is for the Certification Authority to regularly push a certificate revocation list to all clients; but, to reduce the vulnerability window, this requires frequent updates (say daily), which are costly in terms of bandwidth and downtime regardless of whether any impersonation attacks are taking place, and it creates an incentive for attackers to DDOS the CA.

[19] As well as a privacy concern—because a global observer can now link your interactions with the various websites to which your Pico issues revocation requests by their timing and network origin.

[20] Note also that there are often more than two parties involved in a session. I may authenticate Amazon when I buy a book, but the part of the transaction I really care about is the payment via my bank. Unfortunately the current protocol requires Amazon to protect my credit card details so I should check the revocation status of Amazon's certificate. A better protocol would allow me to send an encrypted bundle to Amazon (running the risk that it isn't Amazon) which instructs my bank to pay Amazon (say) £20. Only the bank, not Amazon, should be able to decrypt that bundle.

[21] There is an alternative that is used in many financial networks. The decision to check for revocation might depend upon the value of the transaction. When buying a coffee I might neglect to check revocation status, accepting the risk. When ordering a new television I might insist on checking the status because the value at risk is higher. However this assumes a protocol that prevents the coffee shop from reusing my credentials to buy a television.

Neither of these options is desirable and therefore, in practice, with TLS, invalid certificates are not revoked[22]. New certificates are served but compromised keys have certificates that still appear as valid; therefore impersonation attacks remain possible[23].

4.2 TLS without public key, but with revocation

If we used symmetric key technologies to implement TLS, how would it work? And how would it deal with revocation? We need a Key Distribution Centre, that is to say an authentication server that can do introductions. In the e-commerce context, payment organizations such as Visa or Paypal are third parties known to and (of necessity) trusted by both the user and the website, so we might imagine that they could serve as introducers. Why would they? If the web transaction involved a payment, they might ask for a small percentage. If it didn't, they might ask for a micropayment—perhaps merely the ability to display an advertisement—or, maybe better, they might request a small fee from the website (which probably profits indirectly from the visit even when it does not charge the user). The first time a user visits a website, Visa or equivalent provides an introduction and an initial session key. On subsequent visits, the user and the website already have a shared secret and they can bootstrap further security from that[24].

How could revocation work in this architecture? After the very first contact between client and website via the third-party introduction, the client and the website should negotiate and agree two revocation codes (one each for the client and the website) to be used when needed, as part of an emergency transaction initiated by either party that revokes the current shared key and ideally installs a replacement one. For convenience of exposition, we primarily describe revocation initiated from the website side in what follows.

To set a baseline for comparison, we first outline what occurs in the public key case[25]. The revocation code for the website could consist of $S_{K_w^-}[\text{Revoke}, K_w^+]$ or even just of K_w^- itself. Ideally the original key certificate for K_w^+ would contain

[22] It is interesting to note that browser makers can decide individually on the policy they choose, though they compete against each other on security, features and especially performance.

[23] We don't hear much about such attacks. Is it because they don't happen and we shouldn't worry or because they happen so effectively that we are not aware of them?

[24] This design is architecturally plausible but has the potentially undesirable property that now the client and the website must trust Visa (or equivalent) not just with their money, but also with the confidentiality and authenticity of all their subsequent communications with the website. This problem is shared by all KDS mechanisms; one solution is to combine several keys distributed by mutually mistrusting authentication servers, at the expense of further complexity, latency and potential failure points.

[25] Notation: K^+ is a public key and K^- is the matching private key. A K without exponent is a symmetric key. $E_{K^+}[m]$ or $E_K[m]$ is the encryption of message m under key K^+ or K respectively, whereas $D_{K^-}[m]$ or $S_{K^-}[m]$ is the decryption or

$h(K'^{+}_{w})$, a hash of the replacement public key. Note that website revocation is global, i.e. only one public message is required in order to inform all clients, whereas client key revocation has to be done individually with each website, because of our Pico-specific assumption that websites send the same public key to all their clients, but clients send a different public key to every website.

The revocation transaction does not intrinsically require interaction with any third party, but (in the shadow of a denial of service attack) it does rely upon the website having the ability to post the revocation code securely[26] in a place where the client cannot be prevented from looking, and vice versa.

In the symmetric key case, where the shared secret between client C and website W is K_{cw} after the introduction brokered by the Key Distribution Centre, website W immediately gives client C the string

$$R = E_{K_{cw}}[E_{K_w}[\text{Revoke}, W, K'_{cw}]]$$

where K_w is a master key known only to the website[27]. As soon as the revocation string[28] R has been passed to the client, website and client immediately replace K_{cw} by a new shared secret K_0 (constructed by each of them applying a one-way hash to K_{cw}) and destroy their copies of K_{cw}[29]. The revocation process is for W to publish K_w, which allows the client to obtain K'_{cw} and also provokes the client to cease using all keys based on K_0 and replace them with keys derived from $K'_0 = h(K'_{cw})$. As before, K'_{cw} can be used to share its successor, K''_{cw} encrypted under K'_w, before K'_{cw} is deleted. Note that, just as with the public key case, website revocation is global.

---

signature, respectively, of message m with key K^-. Finally, $h(m)$ is the one-way hash of message m.

[26] The security requirements for posting revocations are surprisingly subtle. Clearly public visibility, persistence, and integrity of the revocation string all need to be assured by the publisher, to prevent the attacker from overwriting it. But what about a denial of service attack where the attacker publishes a bogus revocation string to prevent the correct principal from publishing the real one? We can't require the principal that is revoking their keys to authenticate themselves to the publisher in any conventional way: after all, the keys are being revoked precisely because they may have been stolen. One possibility is for the principal to pre-reserve space with the publisher at an "address" corresponding to a hash of the revocation string, with the agreement that the publisher will allow only the corresponding pre-image to be published at that address. Notice that we needn't care *who* it is that posts the legitimate revocation string: revocation strings are supposed to be kept secret from attackers, so if a revocation string has been stolen then a tamper-evident device has been compromised, and the corresponding keys therefore need to be revoked anyway.

[27] Thus, at that stage, the client cannot decrypt the inner set of brackets $E_{K_w}[\text{Revoke}, W, K'_{cw}]$ which appears as a blob of gibberish.

[28] To avoid confusion: even though we call R a revocation string, this string is necessary but not sufficient to cause a revocation. It behaves like an inert component until activated by primer K_w, mentioned next, which is the trigger for the revocation.

[29] In the client case, after first decrypting R using K_{cw}. Forgetting K_{cw} gives forward security: a subsequent leak of $K_0 = h(K_{cw})$ does not reveal K_{cw}, so even knowing K_w the attacker still cannot obtain K'_{cw} from R.

We need to ensure that the new secret (or private) keys are not themselves compromised by the same attack on secure storage that triggered the revocation[30]. For the sake of conceptual clarity, we shall assume in what follows that the entire website (or at least the authentication and login component) consists of a single tamper-evident device, which may periodically lose confidentiality and/or integrity, and then (following detection of the security breach) be restored to a known secure state on a "new" tamper-evident device prepared (possibly in advance) offline[31]. In practice, a multi-layered security architecture at the website is probably more realistic. For example, we could assume the existence at the website of an even more secure, but less convenient, storage medium that it would not be practicable to involve in routine transactions. In extremis, this medium could even be a USB stick in a safe, or hard copy output from a physically secure logging printer that requires to be typed manually back in as part of the recovery process.

4.3 Avoiding unnecessary re-registration

Remember that the client's first step in establishing a fresh session with the website is to build a leak-proof end-to-end pipe (TLS or its symmetric-key replacement), as described in the previous section. Establishing that this pipe has the correct website at the far end entails an online check by the client that the website has not been compromised since the previous session[32].

The second step (the actual login) is for the website and client to authenticate to one another, and agree a fresh session key. This process requires the client to reveal their identity to the website, and so must take place inside a leak-proof pipe in order to preserve client anonymity. In the public key case, this can be done using protocols such as Sigma-I [7]. How might we implement these steps using only symmetric keys?

Let us suppose that the Pico client Alice has generated a root secret x_n where n is a suitably large number, and computed the reversed Lamport hash chain $x_i = h(x_{i+1})$. The element x_i of this chain is the login credential that will be used in round i to authenticate client Alice to the website[33]. Some time ago, during her initial contact with the website W, Alice privately shared x_0 with W.

[30] In contrast, we don't need to worry about the attacker learning the revocation codes as a result of the attack, but we do need to ensure that the principals can still get access to these codes after the attack has happened.

[31] This is similar to the threat model of Yu and Ryan [12].

[32] This involves checking for the absence of the appropriate revocation string, both in the public key and in the symmetric key case. When a key is revoked, the replacement key is used to share the latest time guaranteed to be before the breach, that is, the latest time at which the tamper-evident hardware was known to be intact. Sessions between then and the new pipe build are assumed to be compromised.

[33] Prover Alice sends the verifier a succession of x_i with increasing i. The verifying website holds x_i from the previous round and cannot derive x_{i+1} from it, but if it receives x_{i+1} it can verify it's genuine by checking whether $h(x_{i+1}) = x_i$. The number n is the length of the chain and determines the number of logins that Alice

It is now several sessions later, and the website currently has the value x_i stored in a table, together with a symmetric key K_i and a randomly chosen number t_i. The value t_i is the current pseudonym for Alice, and is used as a key (in the database sense) to access the authentication table entry corresponding to Alice. This table entry may contain, or link to, Alice's actual (persistent) identity, but need not[34].

In the course of the previous session, Alice pre-agreed the new symmetric key K_i with the website, and this key will be used to establish a leak-proof pipe for negotiating the new login session that is about to be established, thus concealing the "password" x_{i+1} from third parties, including man-in-the-middle attackers. At the time when K_i was agreed, the website W privately gave Alice the revocation string $E_{K_w}[t_i]$ where K_w is a master key known only to W.

Here is the session establishment protocol:

$$(1) \quad A \longrightarrow W : E_{K_w}[t_i]; E_{K_i}[x_{i+1}]$$

W first recovers K_i using t_i, and then verifies that $h(x_{i+1}) = x_i$. W next chooses two random strings t_{i+1} and s_i, then sets $K_{i+1} = h(K_i)$. Now W replies to Alice with the following message:

$$(2) \quad W \longrightarrow A : E_{K_i}[E_{K_w}[t_{i+1}], s_i, x_{i+1}]$$

W replaces x_i with x_{i+1}, t_i with t_{i+1}, K_i with K_{i+1}, and deletes t_i and K_i. The shared keys for the new session between Alice and W are derived from s_i, and are deleted on logout when the session ends.

W can work out which client message (1) is from, by decrypting the first part with K_w, but the attacker can't do this without first breaching the website to obtain K_w. The second part of message (1) can only be decrypted using the key K_i obtained by decrypting the first part. This ensures that only the correct website can obtain the pre-image x_{i+1} of the current authentication credential x_i. The fact that this pre-image is correct authenticates the client to the website, and assures the freshness of both parts of message (1). Message (2) shows knowledge of K_i and x_{i+1}, thus authenticating the website to the client and proving freshness of message (2)[35]. Authentication is thus mutual, however the identity of the client is revealed only to the correct website. Upon receiving the second message, the client simply replaces K_i with $K_{i+1} = h(K_i)$ and deletes K_i.

After the website breach, we cannot prevent the attacker mounting a man-in-the-middle attack to obtain client session credentials to spoof individual sessions, any more than we can prevent this in the public key case. Just as in the public

can perform before having to reload the chain. A version of the Guy Fawkes protocol [13] can be used to refresh x_0 when n is exhausted.

[34] The table entry accessed by t_i will include $h(T_A)$, where T_A is the revocation string for Alice. Alice initiates the revocation protocol with W by revealing T_A.

[35] To allow for the possibility that the two parties might get out of sync, for example through non-malicious communication errors, we might relax the verification slightly. If the pre-image check fails, the verifier should hash again for a configurable number of times. If a match is found, they will have both authenticated and resynchronized.

key case, the credentials for the leak-proof pipe (i.e. $E_{K_w}[t_i]$) will have to change to the fall-back credentials, using the revocation protocol described in section 4.2. But, just as in the public key case, the attacker gains nothing from the client that they can use in the longer term. In particular, once the website is re-secured the client can continue to use the same Lamport hash chain[36] without the risk that the attacker can impersonate her.

We therefore have the property that we wanted to preserve from the public-key scenario, namely that a compromise of the credentials file at the website does not require revocation of the client credentials at that site or any other in order to prevent masquerade attacks subsequent to website recovery[37].

5 Pico without public keys

With Pico, we are much more concerned about offering a viable alternative to remembering and typing passwords than we are about guarding against the cryptanalitic threat of quantum computers. We assume that, at least in the short and medium term, password login on the web will continue to rely on TLS. We are, however, interested in maximizing adoption of Pico. In that spirit, using login credentials that look like passwords rather than public keys is potentially a worthy optimization[38] because it minimizes the changes required of a website to achieve Pico compliance.

5.1 Levels of Pico compliance

We define the following scale of possible levels of Pico compliance, with higher levels promising higher security but requiring greater disruption at the website end and thus greater difficulty for wide-scale adoption and deployment [14].

Level 0. The Pico compatibility mode we presented last year [6] requires no changes whatsoever to the website: the client sees a Pico-friendly login page, rewritten by the Pico lens browser plugin, but the website still receives a traditional username and password and need not even know that a Pico was used to log in. This option would in theory provide maximum deployability, if not for the fact that many websites mess around with Javascript on their login page and as a result cannot be reliably operated upon by a password manager (or by Pico) other than by writing ad-hoc heuristics that cater for special cases.

[36] Note that the hash chains x_i and K_i run in opposite directions. The Lamport hash chain is x_i.

[37] Of course, following a second break-in to the website, the attacker will be able to correlate client identities from the second attack with those stolen from the first. Just as in the public-key case, clients who wish to prevent this will need to update their login credentials (x_0 or K^- in the symmetric and public key cases respectively) before the second attack. However no third-party mediated authentication is required for this.

[38] Hopefully not merely a Needham optimization: replacing something that works with something that almost works but is cheaper.

Level 1. To address this problem we developed the previously-mentioned PMF specification [5]: concise semantic annotations on the login page allow any password manager to submit username and password to the website accurately and reliably, without any guesswork[39]. The website only needs to annotate the HTML of its login page, without changing any of its back-end logic; in return, by becoming password-manager friendly, the website improves its security. There is still the security problem, though, that the password is generated by the user (even though the user no longer needs to remember it) and therefore is probably vulnerable to a guessing attack.

Level 2. To counter that additional problem we specified an extra step in PMF whereby the website allows password-manager-created passwords without imposing its customary composition policy on them (uppercase, lowercase, symbols, digits etc) provided that they are at least t characters long[40], on the basis that only software agents, not humans, will ever use such extravagantly long and hard-to-retype passwords. This strategy allows password managers, including Pico, to generate for each account a distinct random password that will be many orders of magnitude stronger than anything a human could reliably memorize. This strategy requires a small change in the back-end logic of the website (changing the password definition policy to consider any passwords of length t and above as acceptable without checking what classes of characters they contain) but enables security that effectively stops any offline guessing attacks. It is assumed that the website salts and hashes the passwords and that therefore an increase the length of the supplied password does not affect the per-account storage cost of the credentials at the website.

Level 3. Architecturally, from the security viewpoint, it would be preferable not to send the shared secret from client to website at every login but instead to use a challenge-response protocol, as suggested for example by Bonneau [15]. Level 3 would allow for that. While not using public key cryptography per se, it would require changing the back-end logic of the website from verifying a password (salt, then hash, then compare against stored hash) to challenging the client and verifying the response. Challenge-response may add an additional message in the protocol, and thus additional communication delays; with care one might try to optimize away the delays by serving the challenge at the same time as the web page, but this might amount to an even more disruptive change for websites.

Level 4. This level would provide mutual authentication, with the user only supplying their own credentials after having verified the authenticity of the website. In the previous "native Pico" implementation this was achieved with Sigma-I [7], which uses public key cryptography. Using public keys as credentials and a mutual authentication protocol was the original design of Pico [3], but this is clearly the most disruptive solution of all for the website and therefore it is the most damaging in terms of deployability for Pico.

[39] As we said, HTML5 also supports similar annotations.

[40] We suggested $t = 64$ characters taken at random from the `base64` alphabet.

5.2 And when the token is not available?

Unfortunately tamper-evident tokens may be lost or temporarily unavailable. Imagine I receive a phone call while sailing on my yacht. My broker advises me of a business opportunity that requires my digital signature. My signing token is at home in a safe because I didn't want to lose it at sea. There is insufficient time either to collect the token or to deliver a replacement token before the opportunity expires. This suggests that we want a mechanism that temporarily relaxes the requirement for a tamper-evident token.

Fortunately my bank realises this may happen and values my custom. It can establish a proxy service—a remote tamper-evident box—that I may use in such circumstances. Having established my identity by phone they create a temporary account for me on the proxy server which I use to authenticate the transaction. There is a raised level of risk since I am now relying upon the security of my phone which I use to access the proxy server, which is why I revert to using the more secure physical token after I return home.

Since I know that my token is unattended while I am on holiday, I might even ask the bank to disable it temporarily while I am away to reduce the impact if it is stolen.

The research question for Pico is now how to offer this alternative while respecting the primary directive of "you shall not be required to remember any secrets in order to authenticate".

5.3 How should Pico evolve?

The discussion that brought us to write this paper suggests that the security gain from level 3 to level 4 is not significant, and that level 2 is more than strong enough against the threat of offline guessing. Level 2 is also sufficient to offer the desired property that a compromise of the credentials file at the website does not require revocation of the client credentials[41].

We therefore argue that levels 1 and 2 are the sweet spot that offers adequately strong security at acceptable costs in deployability. We won't be pushing for levels 3 or 4 any more: we take level 2 as our new "native mode" for Pico and support levels 0 and 1 as "compatibility mode".

In summary, the Pico login credentials are no longer public/private keys[42] but we continue to use public key technology implicitly insofar as the web relies on TLS.

[41] The website is able to distinguish machine-generated passwords by their length and flag them as such in the hashed credentials file. The website is therefore in a position not to bother those users with a password reset, since their password cannot be realistically brute-forced from the hash even by an arbitrarily powerful offline adversary.

[42] They are instead t-character-long random sequences of `base64` characters. With $t = 64$ they are equivalent to 384-bit symmetric keys.

6 Conclusions

Although our investigation started by looking at whether Pico really needed to rely on public key cryptography to offer its desirable properties, we then broadened the scope to investigate what web authentication might look like if it had to work without public key technologies.

Reviving a discussion that was popular in the last decade of the past millennium, we noted once again that the alleged advantages of public key over symmetric key for authentication don't seem overwhelming when the operational need for revocation is taken into account. We also noted a posteriori that, in the current TLS-based (and thus public-key based) web authentication scenario, revocation essentially doesn't work. Public key may have won on the web in terms of deployment, but hasn't really solved the problem. We argue that, should quantum computers succeed in breaking public key cryptography[43], we could implement alternatives based on symmetric key technology that, though imperfect, would offer comparable benefits and trade-offs when deployed in conjunction with tamper-evident tokens such as Pico.

In particular, one desirable property of a public-key-based authentication system is that users do not need to change their credentials, there or elsewhere, if the website is compromised and its credentials file is stolen. We showed how to achieve this result without using public key primitives.

Coming back to Pico, we had already gradually introduced a compatibility mode that did not require public key cryptography (save for that implicit in TLS). The above investigation prompted us to consider its trade-offs against those of the public-key-based solution. We concluded that, given the ability of the token to generate and define a strong random password for every account (as enabled by PMF), the solution without public key offers comparable security but much greater deployability, because it requires almost no changes on the backend. We have therefore made it our new native mode for Pico.

Acknowledgements

We thank Ross Anderson, Bruno Crispo, Michael Roe and Alf Zugenmaier for their comments on the history of the public key vs symmetric key user authentication debate. Thanks also to Steven Murdoch for his comments on revocation on the web today. The authors with a Cambridge affiliation are grateful to the European Research Council for funding this research through grant StG 307224 (Pico).

References

1. Hern, A.: Did your adobe password leak? now you and 150m others can check. `http://www.theguardian.com/technology/2013/nov/07/adobe-password-leak-can-check` (2013) Accessed: 2015-05-28.

[43] In the sense of allowing an adversary to derive the private key from the public key, for key sizes that are today considered secure against the strongest classical computers.

2. Morris, R., Thompson, K.: Password security: A case history. Commun. ACM **22**(11) (November 1979) 594–597
3. Stajano, F.: Pico: no more passwords! In: Proceedings of the 19th international conference on Security Protocols. SP'11, Berlin, Heidelberg, Springer-Verlag (2011) 49–81
4. Bonneau, J., Preibusch, S.: The password thicket: technical and market failures in human authentication on the web. In: Proceedings of the Ninth Workshop on the Economics of Information Security (WEIS). (June 2010)
5. Stajano, F., Spencer, M., Jenkinson, G.: Password-manager friendly (pmf): Semantic annotations to improve the effectiveness of password managers. In: Proceedings Passwords 2014, Berlin, Heidelberg, Springer-Verlag (2014) To appear. `http://pmfriendly.org`.
6. Stajano, F., Jenkinson, G., Payne, J., Spencer, M., Stafford-Fraser, Q., Warrington, C.: Bootstrapping adoption of the pico password replacement system. In: Security Protocols XXII. Springer (2014) 172–186
7. Krawczyk, H.: Sigma: the 'sign-and-mac' approach to authenticated diffie-hellman and its. In: Use in the IKE Protocols", full version. http://www.ee.technion.ac.il/ hugo/sigma.html
8. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12) (December 1978) 993–999
9. Kohl, J., Neuman, C.: The kerberos network authentication service (v5) (1993)
10. Christianson, B., Crispo, B., Malcolm, J.A.: Public-key crypto-systems using symmetric-key crypto-algorithms. In Christianson, B., Crispo, B., Roe, M., eds.: Security Protocols, 8th International Workshop, Cambridge, UK, April 3-5, 2000, Revised Papers. Volume 2133 of Lecture Notes in Computer Science., Springer (2001) 182–183
11. Myers, M., Ankney, R., Malpani, A., Galperin, S., Adams, C.: Rfc 2560, x. 509 internet public key infrastructure online certificate status protocol-ocsp. Internet Engineering Task Force (1999)
12. Yu, J., Ryan, M.D.: Device attacker models: fact and fiction. In: These proceedings.
13. Anderson, R., Bergadano, F., Crispo, B., Lee, J.H., Manifavas, C., Needham, R.: A new family of authentication protocols (1998)
14. Bonneau, J., Herley, C., Oorschot, P.C.v., Stajano, F.: The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. SP '12, Washington, DC, USA, IEEE Computer Society (2012) 553–567
15. Bonneau, J.: Getting web authentication right: a best-case protocol for the remaining life of passwords. In: 19^{th} International Workshop on Security Protocols. (March 2011)