Authentication at Scale

Eric Grosse and Mayank Upadhyay | Google

Google is investing in authentication using two-step verification via one-time passwords and public-key-based technology to achieve stronger user and device identification.

If bad actors can impersonate you to your service provider, they can do anything you can do. This includes surprisingly destructive behavior, which they can blame on you. If you think you aren't a likely target, remember that attackers might not know you as an individual, but they might just want to leverage millions of high-reputation accounts or leverage your account to get to a real target. So, strong authentication to avoid impersonation is important.

A well-accepted framework for authentication is "something you know" paired with "something you have." In this article, we describe how Google makes such a framework accessible to its diverse user base, what we've learned from working at scale, and some of the directions we're headed.

Account Types

Not all accounts need strong authentication. We divide the spectrum of accounts by value:

- Users might create *throw-away accounts* on the spur of the moment for testing, participating in a pseudo-anonymous conversation thread, or making a one-time purchase without saving payment credentials but with provision for checking order status. Almost any lightweight authentication will do. This area is ripe for innovation but isn't the focus of this article.
- *Routine accounts* are intended to be long-lasting and to protect something of value but do not carry a risk of large financial or reputational loss. An example would be a subscription to an online newspaper. We'd like our authentication methods to be convenient enough to apply to routine accounts. However, some say strong authentication in this case is overkill, and we wouldn't disagree.
- *Spokesperson accounts* are widely understood to represent users. Examples include a blog with a moderately large following or an account at an online store with saved credit card numbers. Some might think that strong authentication is overkill here, too, but we disagree; hijacking of spokesperson accounts is more common than the general public realizes. The consequences of a compromise can range from embarrassment to extensive cleanup costs and financial loss.
- *Sensitive accounts* include an individual's primary email or online banking accounts. Here, loss of data, either by deletion or public exposure, is commonly found to have severe and sometimes unforeseen consequences.
- *Very high-value transaction accounts* are specialized systems used for irrevocable actions such as cross-border monetary flow and weapons release. Such

1540-7993/13/$31.00 © 2013 IEEE Copublished by the IEEE Computer and Reliability Societies January/February 2013

accounts justify stronger protections than are covered in this article.

We focus on protecting access to what we call spokesperson and sensitive accounts. Note that accounts might move from one category to another over time. For example, a store account might be downgraded when its payment credentials expire. Upgrade is more common and less noticeable—for instance, when a Twitter account accumulates more followers or a user registers an email account as a banking account recovery backup.

Common Threats

We contend that security and usability problems are intractable: it's time to give up on elaborate password rules and look for something better. Prominent examples of today's password authentication system failures include the "mugged in Madrid" scam directed against journalist James Fallows' wife;[1] the compromise of Sarah Palin's email account during the 2008 presidential election season;[2] and most recently, the multiple account takeover of journalist Mat Honan.[3]

Security and usability problems are intractable: it's time to give up on elaborate password rules and look for something better.

People are reactive about security; it's rational to invest only as much effort as necessary to reduce risk to an acceptable level. Even with an easier alternative to passwords, justifying transition costs would be difficult. So, we owe it to the reader to not only cite anecdotes but also systematically list the common attacks used in the wild.

Phishing is a widely reported password failure mode: attackers lure users to a login page that looks like one they're used to, perhaps by proxying to the real authentication server or by harvesting their passwords and even supplemental two-factor codes or security questions and answers. By reading about the problem or, even better, hearing about a security breach from friends and family and observing the pain it causes, users improve their chances of recognizing an attack. Password managers can help if they're well integrated with the device and browser, so passwords are used only with the correct sites. However, users would still need to guard against clever attacks.[4]

Reuse is another common password failure mode. A password from a throw-away account at a weakly defended site might be lost through intrusion, then used by attackers to access other, more valuable accounts. Common security advice is to pick a different password for each site. Although this advice is wise, using multiple passwords is burdensome unless combined with a sophisticated password manager. This failure mode remains among the most common preventable problems and was a prime motivation for the two-step verification system we describe later.

A closely related failure mode is offline brute-forcing. Many advise choosing a high-entropy password (http://xkcd.com/936) to harden password hashes, which may be stolen by SQL injection attacks and other means.[5]

Users sometimes type their password in the wrong text field or type a commonly used password rather than the intended one owing to muscle memory. We haven't seen evidence that such lost passwords are actively abused, but internal data on Google employees indicates that such mistakes are common and can leak high-value passwords. Internally, we mitigate this threat by forcing password change when such a mistake occurs, but a better solution would be to use a password manager that understands context and thus can prevent this confusion.

Another authentication failure mode is the use of easily guessed security question and answers, or as wags say, "something you know" paired with "something everyone knows." Even if nobody knows the name of your crush in the third grade, there are a finite number of names to guess. A strong use of security Q&A is to make up random answers, write them down in a safe place, and use them only for account recovery. Such security answers can be thought of as long-term, stable passwords and resist capture by being stored offline and rarely used. Because few users will know to operate in this mode and many would misplace their answers, it's probably better to abandon the security Q&A approach.

Malware infection is another class of failure mode. For example, Zeus logs keystrokes and steals authentication tokens.[6] Although it's important for users to run up-to-date software from trusted sources and some kind of antivirus scanning, these aren't sufficient on their own due to the constantly evolving threat of 0-day attacks that exploit new and unknown bugs in today's complex software systems. Building more hardened platforms is still the best defense, but in this article we also discuss some approaches for channel binding authentication tokens to the client device for which they were issued.

Perhaps most important, password loss can be undetected, only to resurface later on other devices. Therefore, we aim to create consumer-ready tools using hardware-protected public-key cryptography for

both users and devices. Although other projects focus on preventing harm from live malware controlling client machines or from a server break-in, our goal in this article is to prevent persistence. After recovering from an attack, users should be able to regain control of their accounts without losing any long-term credentials.

Because we focus on technical solutions, it's worth acknowledging that some failure modes call for nontechnical solutions. Your roommate knows a lot about you, even when it's safe to use your unlocked computer without getting caught. Although continuity-based face recognition can conceivably help in this situation, technology and cryptography aren't the answer—you need to get a better roommate.

Device-Centric Authorization

Traditionally, user authentication requires that users submit a bearer-token credential, such as a password, to a client device, and the device forwards this on to the server. This model began at a time when users would sit down at a simple terminal connecting to a time-shared computer. Today, the client device is much more capable and might cache credentials for ease of use (for example, a Web browser with a password manager or mail clients that use the Internet Message Access Protocol [IMAP]).

Instead, imagine that each client device has its own strongly asserted identity. When you acquire a new device, you "bless" it with the ability to access your account. This delegation step might require the device to submit multiple factors on your behalf the first time you access an account or require an out-of-band pairing protocol. From then on, the device always asserts its unique credential to the server.

As a result of this delegation, your device (or your well-isolated profile on a shared device) is granted permanent access to your data. If you lose the device, you simply revoke that one instance of delegation; this is clearly less painful than changing your password in the traditional model, which requires reconfiguring all your personal devices. Device-centric authorization also makes abuse detection easier because of the server's ability to distinguish between your multiple devices and to observe their behavior individually.

Today, smartphones follow this model of delegating full account access to a device. The OS is responsible for activity timeouts and protecting the locked screen. Even with a locked screen, the device retains account access so it can receive calendar updates, incoming chats, and so on. Devices are often configured to require a short PIN to unlock the screen. Note that the low-entropy PIN is of no use by itself to an attacker who might have tricked the user into revealing it somehow, much like an ATM PIN is useless without the card.

Figure 1. After successfully typing the right password during sign-in, the user enters a code from a preregistered mobile phone. The user can choose to skip the code next time on this particular browser.

In this model, strong user authentication is applied only when acquiring a new device and when making occasional sensitive transactions, such as setting up email forwarding, deleting all mail, or making a large purchase. Users will forget their passwords if these operations are rare, but that is okay. They should be able to write down their password and store it in a safe place for these rare situations, or have pairing techniques (such as short codes) to bootstrap any new device using an existing device.

Note that this model doesn't necessarily apply to all devices and use cases. For example, users might want a short-term relationship with a device, such as a borrowed machine or kiosk. In this case, the device should offer a guest mode with an obvious session termination gesture that clears credentials and cached data.

Two-Step Verification

As its first large-scale measure for client device authorization, Google introduced an opt-in two-step verification feature, 2sv. As Figure 1 illustrates, first-time users log in to Google from a new computer (after passing the traditional username/password authentication), and then they're asked for a six-digit verification code, which might come from an SMS text message or a voice call to a preregistered phone, an offline application preinstalled on a smartphone, or a one-time "scratch code" from a pregenerated list on their settings page. This code sets a nonexpiring cookie in the browser that makes the user's device a recognized second factor, or a *trusted computer*, for all future authentication.

Users can revoke their trusted computers under the 2sv settings at accounts.google.com/security. Note that this revocation doesn't erase any cached data on disk. We recommend using an encrypted file system and OS-level user separation as a first line of defense against theft.

Our experience with 2sv has been good. Adopted by millions, it's among the largest two-factor authentication deployments in the world. Nearly a quarter million accounts added 2sv during the two days after Mat Honan's story broke, illustrating a phenomenon that we observe more broadly: people take security more seriously after an acquaintance or public figure has suffered

harm. After studying hijacking campaigns directed at high government officials, we found that among the hundreds whose password had been stolen, presumably by phishing, two officials had enabled 2sv and were successfully protected from compromise.[7]

However, not nearly enough of our users are protected, and we recognize that awkward corner cases and inadequate documentation contribute to this.[8] We will continue to polish the rough spots. To minimize setup time, we encourage SMS or voice delivery of 2sv codes. Approximately 10 percent of our users subsequently install and provision the offline smartphone application for code generation, which doesn't require working cellular service or even a registered phone number.

> **In the future, we envision users will own enough authorized devices that they can always use an old device to authorize a new one.**

When deployed at scale, some users will experience account lockout owing to lack of coverage while traveling, temporarily slow text message delivery, loss of the device, changing of mobile phones without requesting phone number portability, and so forth. We find that customer support for account recovery is crucial in wide 2sv deployment.

Fortunately, many users set up backup modes for code generation, such as home or work landlines, a family member or friend's phone, and paper-based codes. These users tend to self-recover from issues related to loss of their primary 2sv code generator. We've also found that the smartphone app users rarely need additional help because, among other reasons, they're unaffected by message delivery issues.

Users with many client-side applications that allow for only traditional username/password-based sign-in tend to have the hardest time setting up 2sv. Typical examples include IMAP-based mail clients on desktops and certain smartphones. To allow backward compatibility on those apps and devices, we provide a transition feature called application-specific password (ASP). An ASP is intended to be a high-entropy machine-generated password that's hard to remember and consequently hard to phish. Unfortunately, the same properties that increase such passwords' security also cause friction for users. To fix this problem across the industry, we prefer that client platforms employ a centralized account management model with a browser sign-in option, as the Android OS does. Another weakness of ASP is the misimpression that it provides application-limited rather than full-scope account access. (OAuth, which we discuss later, is the right tool for that job.)

In Android OS versions Ice Cream Sandwich and higher, 2sv users can set up their phones via a browser-based sign-in flow that the system offers when a second factor is necessary. The browser flow enables a flexible HTML-based UI that incorporates a 2sv challenge, avoiding the need for ASP. Furthermore, Android's centralized account management model makes it unnecessary for multiple apps to ask the user for the same password and 2sv code; instead, these apps request the system account manager for short-lived scoped tokens for the data they need to access. Therefore, users in the Android ecosystem have an easier time setting up 2sv.

Initially, we thought of 2sv as part of user authentication, much like the one-time password (OTP) tokens that enterprises commonly require for remote authentication. To make 2sv practical for consumers, we reduced the default verification requirement to once per month. But, we found that 30 days is either too short or too long; it's annoyingly frequent and disconcerting when applied independently to every browser in every device and yet too large a window of vulnerability for a lost, unlocked device.

We changed our mental model to treat 2sv primarily as a means of permanently authorizing a client device. (Users can still achieve the old behavior if desired; the 2sv validation page includes a checkbox that, if not checked, indicates the 2sv cookie should expire at the end of the browser session rather than last forever.)

Requiring verification once per month had a training advantage; verification was frequent enough to remind users to bring their phone when traveling or update their registered phone number after a change. Now if users go a long time without typing a 2sv validation code, we might remind them about their 2sv enrollment and phone number information and perhaps even ask for a practice code. But we won't lock them out of their account if the browser already has a valid 2sv cookie. We're reasonably satisfied with this balance.

A final interesting observation about 2sv is that it's abused by account hijackers. After stealing the account password and breaking in to the account, hijackers add 2sv (with their own phone number) just to slow down account recovery by the true owner! Anecdotally, we've heard that the online game World of Warcraft—one of the few other consumer services that has very widely deployed two-factor authorization—has seen the same phenomenon.

From a security perspective, 2sv is effective against the common failure modes of reused passwords and lost password hashes, but in the long term, it will be ineffective against clever phishing. It won't protect users against someone who steals both their passwords and phone. We're aware of two targeted account hijackings getting past 2sv: in one, the attackers allegedly used social engineering against the phone network, resulting in loss of SMS; in the other, they allegedly used malware on the phone, resulting in loss of voicemail.

Smartcard-Like USB Token

With the current version of 2sv, users type a code into a new device to authorize it. To better protect against phishing and, at the same time, to make the server side immune to authentication database theft, we're interested in smartcard-like solutions based on asymmetric or public-key cryptography.

Others have tried similar approaches but achieved little success in the consumer world. Although we recognize that our initiative will likewise remain speculative until we've proven large-scale acceptance, we're eager to test it with other websites, following three guiding principles:

- For maximum portability, this method mustn't require software installation on the host other than a compliant Web browser.
- One device should be sufficient with a reasonable number of websites for which users have accounts. But, for privacy preservation, the websites mustn't be able to correlate users based on the device.
- User device registration with target websites should be simple and shouldn't require a relationship with Google or any other third party. The registration and authentication protocols must be open and free for anyone to implement in a browser, device, or website.

Our first implementation of such a solution has been an experimental USB token for 2sv. The token speaks on USB without needing special operating system device drivers. A higher-level protocol specifies packet formats for obtaining signed assertions from this token and can be exercised by application-level code on the host OS. The USB token also has a capacitive touch-sensitive area for user confirmation.

We're currently working on an internal pilot of this 2sv token to validate the form factor and user experience. Consumer provisioning should allow users to buy a compliant token from a vendor of their choice, insert it into a computer where they're already authenticated to a website, and register their token with a single mouse click.

A compliant browser will make two new APIs available to the website to be passed down to the attached hardware. One of these APIs is called during the registration step, causing the hardware to generate a new public-private key pair and send the public key back to the website. The website calls the second API during authentication to deliver a challenge to the hardware and return the signed response. The protocol specification calls for a key-generation process inside a secure element with attestation and for the private key to never be exposed outside.

Besides the physical controls we mentioned, additional privacy protections are built in to the 2sv token. Note that the secure element never returns a previously generated public key in any new registration step. This makes tracking users across websites difficult by using the token as a supercookie that bypasses other anonymizing precautions. Furthermore, the browser provides the token with a hashed identifier of any website requesting a signing operation. This allows the token to withstand tracking attempts in which a website shares the registered user's public key with another website as well as datacenter intrusion attacks in which public keys are stolen. Finally, the protocol allows for extensions for the website to send a display string to the user, which we will use for anti-malware mitigation.

We recognize that multiple form factors are necessary for broad consumer adoption. A removable USB-based token with lean and audited firmware has a small attack surface for malware and a clear mental model for privacy. However, having to carry an additional token is likely to be a barrier to adoption for many consumers. Some more appealing form factors might involve integration with smartphones or jewelry that users are likely to carry anyway. We'd like your smartphone or smartcard-embedded finger ring to authorize a new computer via a tap on the computer, even in situations in which your phone might be without cellular connectivity. For many users, travel is likely to present the need to sign in to a new machine. We're finding that the biggest technological challenge isn't cryptography but the lack of a standardized interface on consumer platforms for device-to-device interaction in the real world. Some technologies with which we're experimenting are unsecured radio frequency communication (RFCOMM) (unpaired Bluetooth) and near-field communication (NFC).

In the future, we envision users will own enough authorized devices that they can always use an old device to authorize a new one and will only need a strong password for deep backup. As the overall authentication system strengthens, we predict that authorization flow will be the next avenue of attack, so we seek to harden this process in advance.

Channel Bindings

We have been focusing on how clients authenticate to the server because authentication in the other direction is settled: Secure Sockets Layer (SSL) with server certificates. Let's see now how SSL can harden client authentication.

Entering a username and password in a Web login page for user authentication, or a 2sv code for client device authentication, sets a cookie in the browser's local storage. Although browsers provide various defenses against rogue JavaScript stealing these cookies, they're typically less protected than the private keys associated with SSL client certificates, which can be stored in the OS's keychain or even under hardware protection, such as a Trusted Platform Module (TPM) or smartcard. This leads to the idea of using client certificates and SSL session secrets instead of cookies. Although client certificates have been around since the early days of the Web, they never became popular because the user interface for adding them was painful, often involving complicated sequences of browser- and platform-specific popup screens with no website customization. Using a single client certificate is a privacy mistake because it enables tracking; on the other hand, using multiple client certificates and asking users to select one manually is a burdensome user experience. Given these disadvantages, the consequent widespread use of cookies, and the amount of application software that would need to be updated, the idea of switching to client certificates seems infeasible.

> **ChannelID in Chrome browser version 24 is being deployed with no user-visible effect. It's just a silent hardening of the platform, an unusually pleasant way to roll out new security features.**

However, a new approach shows great promise—binding cookies cryptographically and automatically to the SSL client. The first time a compliant browser talks to a new domain, it automatically generates a key pair, which is reused for future SSL connections to that domain. Cookies (or other bearer tokens) can be bound to that client key so that they're usable only inside connections that the client initiates.[9]

We used a fast public-key cryptosystem, elliptic curve P256, and used the public key directly rather than a self-signed X.509 certificate. The computation and network overhead was low, even with the new TCP connections. In a typical serving architecture, the SSL terminator on a front-end server passes the client public key (or its hash) to the back-end server. Back-end servers can adopt channel bindings incrementally whenever they're ready for the change.

This feature, called ChannelID in Chrome browser version 24, is being deployed with no user-visible effect. It's just a silent hardening of the platform, an unusually pleasant way to roll out new security features. In keeping with the zero-user-interface design, deleting browser history or cookies and site data will automatically delete the corresponding domain key pairs. Moreover, alternative browser profiles and incognito mode use different key pairs, just as they use different cookies and site data.

Using the TPM chip built in to many laptops for hardware-protected cryptography is another appealing way to protect these private keys and limit loss from malware or disk imaging. We hope our efforts will encourage hardware vendors to more widely include higher-performance TPMs that could be employed for this purpose. In the meantime, we plan to experiment with the smartcard-like USB token as a semi-permanent secure element in computers. Such a USB token becomes a kind of "ignition key" that locks a user's computer as soon as it's removed.

Server-Side Technology

For server authentication by the client device, the situation is better. Most browsers can verify SSL certificates properly, and sites can turn on features such as HTTP Strict Transport Security to prevent downgrade attacks. Although we must remain vigilant for SSL protocol and implementation mistakes—and server key compromise—the largest observed risk of man-in-the-middle attacks on SSL is the compromise of root certificate authorities, such as Diginotar. Even exotic attacks grow in importance over time, so we've focused substantial effort on certificate transparency and related ideas.

Certificate transparency ensures that server certificates are published in a few well-known locations, so a website operator can verify that it holds the only certificates that can authenticate as its servers. A browser that receives a server certificate gets cryptographic proof that it's been published as well as supplemental processes to catch unreliable publishers, attackers who compromise the central systems, or even a government coercing the central players. Because correct operation of the logs can be verified independently, this scheme doesn't introduce yet another trusted party.[10]

Risk analysis is often left out of authentication discussions because it's invisible to the user, but it's an important part of the system. For average users with

weak or reused passwords, this back-end risk-based checking is particularly critical to reduce what would otherwise be widespread account hijacking. If 2sv and other two-factor systems comprise "something you know" and "something you have," this might be called "somewhere you are" and "some way you behave." A geolocation pattern of login IP addresses that is violated suddenly should trigger extra concern. The server might post an alert, as Gmail's Web interface does when detecting an unusual country of login. As with credit card risk analysis, it must allow for people going on vacation. In more extreme risk signal cases, forcing users to answer additional questions to verify identity might be justified. Some of these risk signals are sophisticated, going well beyond login geolocation to include users' behavior after they've logged in. It can be difficult to design these notification and challenge systems to work effectively without creating extra opportunities for phishing attacks by mimicry.

A server can adopt a federated login approach, effectively letting one server pass the burden of validating user authentication to another server using browser redirection. This is especially appealing for small websites, which can leverage large sites' much richer set of authentication and risk analysis technologies to overcome new users' reluctance to create and manage yet another account.

Service Accounts and Delegation

Trying to eliminate passwords in the real world revealed two important aspects that are worth mentioning: authentication by applications and account sharing.

Programs like the Secure Shell (SSH) client or a Web browser executing RSA operations speak directly on behalf of a person. Other programs, such as print servers, also need strong identity yet have existence independent of any person.

Authenticating programs differs from authenticating people in some important ways. For instance, there's often no good place to store a credential. Passwords certainly shouldn't be hardwired into the source code or in a command invocation line; most developers have learned to avoid these rookie security mistakes. But storing them in a configuration file is problematic; how do we control access to the file from an unauthenticated program? And if the program is authenticated, why does it need to read a credential? Do we update the password every time developer team membership changes?

Cloud computing "service accounts" are a modern solution to this problem. We can rely on the cloud infrastructure to testify about a program's identity to other components or even outside systems. Think of the cloud infrastructure as holding the equivalent of a smartcard with a private key that it uses on behalf of the program. For instance, consider the case of applications running on Google App Engine (GAE). There are three common design patterns:

- GAE provides some built-in sensitive resources, such as the Datastore. When an application gets a handle for talking to the Datastore, it comes with implicit authentication. The application can be confident that any data it puts in the Datastore is accessible only to a future instance of itself and not to other apps.
- GAE enables the app to reach other resources participating in the OAuth authentication ecosystem we describe later. For example, the GAE app named 1234567@appspot.gserviceaccount.com can acquire an OAuth token valid for one hour, scoped to the Google Docs API.
- GAE participates in lower-level handshakes, allowing an app to talk to proprietary architectures. Assume that an app has to authenticate itself to a particular bank's gateway, which mediates access to various other resources. The app can request GAE to sign a blob with the RSA private key that GAE manages for it. The app uses that token to authenticate to the bank gateway, which replies with a bank-specific credential that the app can use to access other resources directly. In this pattern, GAE automatically manages the app's public key, and the bank obtains it from a well-known discovery endpoint.

Our final topic, delegation, often applies to these service accounts but also meets real-world personal needs, as we learned when we began actively fighting password sharing in our organization. Generally, delegation refers to an account owner granting a third party scoped access to the account, possibly involving restrictions to certain objects or actions. In the narrowest case, the delegated scope might include only knowledge of the account's email address, so delegation is one implementation of federated login.

The best delegation systems might be the ones tightly integrated with an application. For example, email account owner Alice designates deputy Bob by explicitly authorizing the deputy's account bob@example.com to read Alice's incoming messages and to send messages with an authenticated address such as "From: Alice <alice@example.com> (sent by bob@example.com)."

Such integrated application behavior can provide productive sharing without granting full account access. Alice might trust Bob to read and send email but not to approve payments over some threshold.

Adding delegation features to each application independently would require a lot of implementation effort

and likely lead to gratuitously different systems. Web service providers searched for a substitute to the bad practice of users giving away their passwords to third parties for scraping information like contacts, and came up with multiple independent protocols like Google's AuthSub, Yahoo's BBAuth, and Facebook's Login. The industry has recently made great progress toward a unified standard under the OAuth 2.0 umbrella.

OAuth provides a way to grant scoped access to an account using a bearer token inside SSL, which the account owner can revoke on a per-delegation basis.[11] As a potential improvement, we envision that OAuth bearer tokens could be channel-bound to an SSL session that uses client authentication.

An aside on terminology: there is another authentication standard called OATH that has nothing to do with OAuth. The Google Authenticator App for Android, Blackberry, and iOS implements the HMAC-based OTP (HOTP) algorithm (RFC 4226) and the Time-based OTP (TOTP) algorithm (RFC 6238), which are central to OATH. It's easy to get confused by the proliferation of labels.

The Google Cloud Print architecture provides a nice example of both service accounts and delegation. When printing a document, users share limited-time read access to that one document with the service account embedded in the cloud-ready printer. The service account identifies itself with an OAuth2 refresh token obtained using a version of the OAuth2 device flow. Users don't need to grant the printer any more access to their personal information than the contents of the document. Conversely, the printer (which might be in a public location) doesn't give users direct connectivity or authority.

Along with many in the industry, we feel passwords and simple bearer tokens, such as cookies, are no longer sufficient to keep users safe. We're investing in these client-side technologies and authentication methods using one-time passwords and public-key-based technology to strengthen user and device authentication. ■

Acknowledgments

We thank Ben Lauri, Brian Eaton, Diana Smetters, Dirk Balfanz, Eric Sachs, Frank Cusack, Marc Donner, Marius Schilder, Naveen Agarwal, Nishit Shah, Roberto Ortizr, Sam Srinivas, Úlfar Erlingsson, the rest of the 2sv team, and the anonymous referees for helpful comments and material.

References

1. J. Fallows, "Hacked!," *The Atlantic*, Nov. 2011; www.theatlantic.com/magazine/archive/2011/11/hacked/308673.
2. "Tennessee Man Convicted of Illegally Accessing Sarah Palin's E-mail Account and Obstruction of Justice," Dept. Justice, 30 Apr. 2010; www.justice.gov/opa/pr/2010/April/10-crm-509.html.
3. M. Honan, "How Apple and Amazon Security Flaws Led to My Epic Hacking," *Wired*, 6 Aug. 2012; www.wired.com/gadgetlab/2012/08/apple-amazon-mat-honan-hacking.
4. K. Bhargavan and A. Delignat-Lavaud, "Web-Based Attacks on Host-Proof Encrypted Storage," *Workshop Offensive Technologies* (WOOT 12), Usenix, 2012; http://moscova.inria.fr/~karthik/pubs/host_proof_woot12.pdf.
5. F. Pesce, "Lessons Learned from Cracking 2 Million LinkedIn Passwords," Qualys Security Labs, 8 June 2012; https://community.qualys.com/blogs/securitylabs/2012/06/08/lessons-learned-from-cracking-2-million-linkedin-passwords.
6. K. Stevens and D. Jackson, "Zeus Banking Trojan Report," Dell SecureWorks, 11 Mar. 2010; www.secureworks.com/research/threats/zeus.
7. "Ensuring Your Information Is Safe Online," Google Official Blog, 1 June 2011; http://googleblog.blogspot.com/2011/06/ensuring-your-information-is-safe.html.
8. J. Fallows, "Gmail's 2-Step Verifications: Some FAQs," *The Atlantic*, 9 Aug. 2012; www.theatlantic.com/technology/archive/2012/08/gmails-2-step-verification-some-faqs/260934.
9. M. Dietz et al., "Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web," *Usenix Security Symp.*, Usenix, 2012; https://www.usenix.org/conference/usenixsecurity12/origin-bound-certificates-fresh-approach-strong-client-authentication.
10. B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," Internet Engineering Task Force, 29 Nov. 2012; http://tools.ietf.org/html/draft-laurie-pki-sunlight.
11. D. Hardt, "The OAuth 2.0 Authorization Framework," Internet Engineering Task Force, 31 July 2012; http://tools.ietf.org/html/draft-ietf-oauth-v2.

Eric Grosse is vice president of security engineering at Google. His research interests include all areas of practical computer and network security and privacy. Grosse received a PhD in computer science from Stanford. He's a member of ACM, IEEE, and SIAM. Contact him at ehg@google.com.

Mayank Upadhyay is principal engineer at Google. His research interests include many aspects of Web security, wireless network security, and usability. Upadhyay has an MS in computer science from Stanford. Contact him at mayank@google.com.