acmqueue

Certificate Transparency

Public, verifiable, append-only logs

Ben Laurie, Google

On August 28, 2011, a mis-issued wildcard HTTPS certificate for google.com was used to conduct a man-in-the-middle attack against multiple users in Iran. The certificate had been issued by a Dutch CA (certificate authority) known as DigiNotar, a subsidiary of VASCO Data Security International. Later analysis showed that DigiNotar had been aware of the breach in its systems for more than a month—since at least July 19. It also showed that at least 531 fraudulent certificates had been issued. The final count may never be known, since DigiNotar did not have records of all the mis-issued certificates. On September 20, 2011, DigiNotar was declared bankrupt.

The damage caused by this breach was not confined to Iran. When the DigiNotar roots were eventually revoked, two weeks after the initial discovery, they included one used by the Dutch government to provide Internet services. This revocation prevented the Dutch from buying and selling cars, electronically clearing customs, and purchasing electricity on the international market, among many other things. Also, of course, every Web server with a certificate issued by DigiNotar had to scramble to get a new certificate.

This was not the first time a CA had been compromised, nor was it the last. The Comodo Group (fraudulent certificates issued for Google, Yahoo!, Skype, and others),[1] TürkTrust (unauthorized google.com certificate),[2] and ANSSI (certificates issued via an intermediate allegedly for local network monitoring) have all had reported breaches or internal mis-issuance. More such incidents are sure to come.

It is easy to blame the CAs' poor security for these breaches, but the fact is that the state of the art in software engineering and system design cannot render anyone absolutely safe from attack. While the CAs surely must take some of the blame (particularly those that know what has happened but keep it quiet in the hope they will get away with it!), no one yet knows how to build a completely foolproof system. So, how can we make things better?

ALTERNATIVES TO THE CA

Perhaps it is instructive to take a step back and consider the problem we're really trying to solve here: ultimately, we want to ensure that Web users are actually talking to whom they think they're talking to, and that no one else can intercept the conversation. That's really an impossible goal—how can a computer know what the user is thinking—but for now let's reduce the problem to a slightly different one: how to ensure the Web user is talking to the owner of the domain corresponding to the URL being used. Granted, this is a rather weak strategy, but it does work in at least some circumstances. Most people know the right URL for the big Web sites such as Google, PayPal, and eBay. Likewise, if users follow a correct link from an honest source (which is, in practice, most links), they are protected.

The solution the computer world has relied on for many years is to introduce into the system

trusted third parties (CAs) that vouch for the binding between the domain name and the private key. The problem is that we've managed to bless several hundred of these supposedly trusted parties, any of which can vouch for any domain name. Every now and then, one of them gets it wrong, sometimes spectacularly.

What are the alternatives? And what are the constraints on them? Let's talk about constraints first.

CONSTRAINTS

The constraints will definitely vary depending on the nature of the system. Here the focus is on the Web from the point of view of Google Chrome. The following are some constraints to consider in devising a secure system for Chrome:

- **Migration path.** There must be some plausible way to get there from here. Solutions that require the whole world to change on a flag day, for example, are just not going to fly.
- **Generally applicable.** No one is special. Everyone must be able to participate. In other words, the solution must scale.
- **(Almost) no added latency.** Chrome is very passionate about page load times, so they cannot be made noticeably slower. A corollary is no synchronous out-of-band communications: everything needed before a page loads must arrive in the same channel as the page itself. Experience shows that if we have to consult another source, it will fail and it will be slow. (For a discussion of that experience, see https://www.imperialviolet.org/2014/04/19/revchecking.html.)
- **Don't make the situation worse.** For example, "fixing" one set of trusted third parties by introducing another doesn't seem like a step forward.
- **Don't push the decision onto the end user.** We already know users don't understand certificate warnings. Expecting them to make sense of even more complex schemes is unlikely to be the answer.

Given the shortcomings of CAs and the system of trusted third parties to ensure Internet security, several alternatives have surfaced. At Google we have found only one (spoiler alert), Certificate Transparency, that can overcome all the constraints and present a reasonable solution to the security problem. Before considering why that's the case, let's look at some of the other alternatives.

PINNING

One alternative is for Web sites to advertise which certificates (or CAs) are correct for them, with the browser rejecting any certificate that is not on that list. Right now, pins for some sites are built into Chrome, but proposals exist to allow anyone to advertise a pin (for example, https://datatracker.ietf.org/doc/draft-ietf-websec-key-pinning/ and http://datatracker.ietf.org/doc/draft-perrin-tls-tack/).

Pinning fails for subtle reasons. What happens when something goes wrong? Clearly a pin can't simply be replaced by a new pin, or the whole point is defeated. So, pins expire after some preset time. If you lose your key before the pin expires, then your site doesn't work until it does expire. Short expiration times provide little protection from interlopers, but long pin times could mean some serious downtime in the event of disaster.

Right now, if this happens, the recourse is to contact Chrome and ask to change your pin, but this is not a scalable, inclusive solution. Furthermore, at least while pins are built into Chrome, pinning introduces a new trusted third party (i.e., Google).

NOTARIES

Another popular alternative is to use notaries. The best-known notaries are the Perspectives project (http://perspectives-project.org/) and Convergence (http://convergence.io/). Google also ran one for a while, called the SSL Certificate Catalog,[4] but decided not to continue to support it after starting work on Certificate Transparency. The idea is to scan the Internet periodically, ideally from multiple points of view, inserting all certificates into a notary log, which can later be asked, "Have you seen this certificate before?" If the answer is "no" or "not very often," then the certificate might be viewed with some suspicion.

This idea has quite a few problems. The biggest one is that an answer of "no" could simply indicate that the site has just changed its certificate. Should sites be inaccessible every time they renew their certificates? That seems unwieldy. Second, the notary approach involves an out-of-band check, which breaks one of the deployability rules. Third, a bold attacker might deploy a fake certificate widely, which would make the notaries think everything is fine. Finally, it introduces a new trusted third party.

DNSSEC

Another alternative is to base trust in DNSSEC (Domain Name System Security Extensions). Two mechanisms exist for this purpose: DANE (DNS-based Authentication of Named Entities; https://tools.ietf.org/html/rfc6698) and CAA (Certification Authority Authorization; https://tools.ietf.org/html/rfc6844). Strictly speaking, CAA makes DNSSEC optional but strongly recommended. Both DANE and CAA associate particular certificates or CAs with hosts in the obvious way by having appropriate DNS records for the host's name. The difference is the way the records are used: DANE records are to be checked by clients when connecting to servers; CAA records are to be checked by CAs when issuing certificates. If the CA is not included in the CAA record for the new certificate, then it should refuse to issue.

Both proposals have problems that are inherent in DNSSEC, but CAA also has the further issue of not really solving part of the problem—namely, mis-issuances by subverted or malicious CAs. Clearly they will simply not bother to consult the CAA record.

More importantly, however, DNSSEC, like the existing PKI (public-key infrastructure), introduces trusted third parties into the equation, which may or may not fulfill their duties. The trusted third parties in this case are DNS registries and registrars. Sadly, their security record is substantially less impressive than that of CAs.

Some argue that DNSSEC has an inherent protection against subversion of these trusted third parties, because the public nature of DNS makes any meddling immediately visible. The problem with this theory is that DNS is a distributed system—my view is not your view, and there's nothing ensuring that our two views are consistent. It is thus relatively easy for attackers to present a split world showing one set of DNS records to their victims and another set to those who seek to check the integrity of DNS.

Another problem is simply that DNSSEC so far is not widely deployed, so we would be gating one improvement on another for which we have already waited well over a decade (indeed, I was busy fixing the "last" problem in DNSSEC more than eight years ago [see RFC 5155 http://tools.ietf.org/html/rfc5155]).

Finally, experiments indicate that at least four percent of clients can't get DNSSEC records at all because of routers that take over the DNS and do not support DNSSEC, because of captive portals and the like, and because of blocking TCP on port 53 (DNS is usually served over UDP, but larger records require fallback to TCP), and other causes.

Note, however, that DANE would be useful in the context of SMTP. SMTP servers are already fully at the mercy of DNS and currently use only opportunistic TLS (Transport Layer Security). DANE would definitely be an improvement.

BITCOIN-BASED SOLUTIONS

I have written extensively on what is wrong with Bitcoin (for example, it is the least green invention ever, and all of its history could be destroyed by a sufficiently powerful adversary if it were truly decentralized, which it is not). Nevertheless, people continue to worship irrationally at the altar of Bitcoin, and this worship extends to DNS and keys—for example, DNSChain (https://github.com/okTurtles/dnschain).

Apart from being an extremely costly solution (in terms of wasted energy, in perpetuity), it also introduces new trusted third parties (those who establish the "consensus" in the block chain) and has no mechanism for verification.

CERTIFICATE TRANSPARENCY

The drawbacks of all these alternatives have led us at Google to pursue a different approach, called Certificate Transparency. The core idea behind Certificate Transparency is the public, verifiable, append-only log. Creating a log of all certificates issued that does not need to be trusted because it is cryptographically verifiable (and it turns out this is possible, as explained in more detail later) allows clients to check that certificates are in the log, and servers can monitor the log for misissued certificates. If clients decline to connect to sites whose certificates have not been logged, then this is a complete solution. It becomes impossible to misissue a certificate without detection.

This mechanism allows us to meet all the constraints listed earlier. There is a migration path: certificates continue to be issued and revoked as they always have been, and over time more and more clients check for log inclusion, and more and more servers monitor the logs. Even before all clients check, the fact that some do confers a herd immunity on the remainder.

Everyone can participate. It is not hard to get a certificate into a log, and since the log itself makes no judgment on the correctness of the certificate, there's no change to the revocation of bad certificates, which is still done by the CAs.

Latency is not added because log-inclusion proofs are compact and included in the TLS handshake.

No trusted third party is introduced. Although the log is indeed a third party, it is not trusted; anyone can check its correct operation and, if it misbehaves, prove that it did.

Finally, Certificate Transparency does not push the decision onto the user. The certificate is either logged or it is not. If it is logged, then the corresponding server operator (or other interested parties) can see it and take appropriate action if it is not valid. If it is not logged, then the browser simply declines to make the connection.

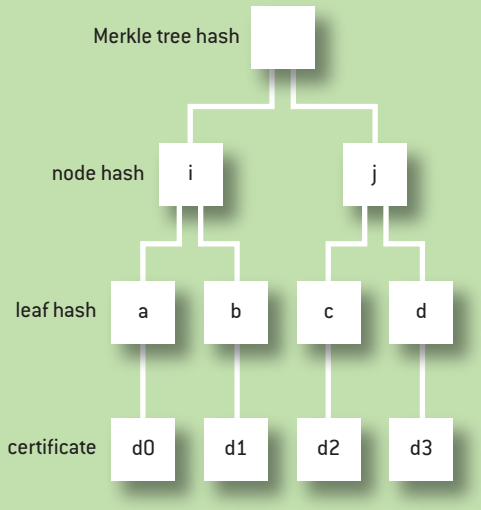
HOW IT WORKS

How do you build a verifiable, append-only log? This turns out to be relatively straightforward in essence, although there are some design tradeoffs and some interesting problems to overcome. One obvious approach would be for clients of the log simply to verify, by periodically downloading the entire log, that it satisfies the append-only property. A client could also compare its copy of the log with other clients' copies to verify that they are all seeing the same log. Then everyone would know that the log is indeed public and append-only.

This is very wasteful of bandwidth and storage, however. A better approach is to use Merkle trees, as shown in figure 1. A Merkle tree's leaves are the items in the log. Each branch in the tree is a cryptographic hash of the nodes below the branch (this follows the somewhat odd convention that trees grow downward; the root is at the top, the leaves are at the bottom). Clearly, from the properties of cryptographic hashes, each node is a summary of all the nodes below it: none can be modified without changing the hash. Thus, the root of the tree is a summary of all of the leaves. This means it is now possible for clients efficiently to verify that they have seen the same tree by simply comparing a hash.

That is not all we want from a log, however. We would like to verify that things that are claimed to be in the log are actually in the log and that the log has the append-only property (or to put it another way, the entirety of yesterday's log is included in today's log). Fortunately, a Merkle tree is an efficient way of accomplishing this. To show that a particular leaf is in a log, all that is needed is the hash of that leaf and a list of neighboring hashes working up the tree. The root can be recalculated from this list, and, because the hash is cryptographic, this means the leaf must be present in the tree (otherwise, it would not be possible to produce a list of hashes that combine with the leaf hash to produce the root hash). Similarly, yesterday's root can be linked to today's root by a list of hashes representing the new items added to the tree.

FIGURE 1 **Merkle Hash Tree**



One more ingredient is needed for a complete system. If a log misbehaves, then we need to be able to prove it has done so. Luckily, this is again rather easy. The log simply needs to sign things. In principle, in fact, the log could sign just one thing: the root of the tree. This is also a signature for everything included in the tree.

This leads to the first tradeoff. Logs should be both highly available and consistent. In order for TLS clients to ensure that certificates are actually in the log, each certificate must include some kind of proof of inclusion. (Technically, the proof can come anywhere in the conversation with the server, but since rolling out new server software will take many years, the only option that can be universally deployed right now is to include the proof directly in the certificate.)

Our initial thought was to include a Merkle proof—that is, a signed tree head—and the hashes leading from the entry for the certificate to the signed head. This runs sharply into the available/consistent tradeoff, however: in order to make the log highly available, you need to run multiple instances that are geographically distributed. That means the process of synchronizing the instances can potentially take some time, and they must be synchronized to achieve the append-only property. Therefore, before a CA can issue a certificate, it must first submit it to the log and wait until the log has synchronized all its instances (in order to assign the new certificate a position in the log) and returned a Merkle proof for the new version of the log. Although this can be quite fast most of the time, in the face of network and server outages it could take quite some time—hours or possibly even days.

CAs found this delay in their issuance pipelines to be unacceptable. In fact, there are points on the tradeoff spectrum that are even slower—for example, the log could return both a proof of inclusion and proof that log monitors had not only seen the new certificate, but also had time to take action if it had been incorrectly issued.

Luckily, there are also points on the spectrum that are faster. The version that was finally implemented has the log server return a signed timestamp for the new certificate, known as an SCT (signed certificate timestamp). This signed timestamp is a promise for future inclusion in the log. Each log has an MMD (maximum merge delay) parameter, which says how long after issuing the signed timestamp it is permitted to wait before inclusion.

On the face of it, this would seem to allow the log to respond instantly, since all it needs to do is generate a signature, a process that takes under a millisecond on modern hardware. But this is not quite true. Since the SCT is a promise to include the certificate, the log absolutely cannot afford to lose the certificate once it has issued an SCT. Therefore, the log must have some redundancy in the storage of incoming certificates; the certificate should probably be stored in multiple data centers. This does allow the consistency requirement to be relaxed. A log can store incoming certificates in some subset of its redundant instances and later resolve any inconsistencies before deciding on an order for new certificates and generating a new version of the log. This kind of redundancy is substantially faster than consistent redundancy.

The next tradeoff is MMD. Obviously, those monitoring logs would like MMD to be as short as possible, since a log colluding in deliberate mis-issuance will presumably delay logging for as long as possible. Note that if the log does not collude or lose control of its key, the CA (or the CA's attacker) cannot influence the time taken by the log to integrate new certificates. Log operators, on the other hand, need the MMD to be long enough to allow logs to establish a consistent state, perhaps even in the face of software errors. We are still deciding what an acceptable MMD is, but clearly it has to be at least hours and possibly as long as a couple of days.

The third tradeoff is the number of logs that each certificate should be logged in. There are two reasons for favoring the use of more logs. First, if a log does go bad, clients will stop trusting it. If all the SCTs for a certificate are from bad logs, then the certificate will no longer work. Second, the more SCTs are required in a certificate, the harder it is for an attacker to avoid detection, since the attacker would have to control both the CA (or the CA's key) and all the required logs (or their keys).

Our current thinking is that each certificate should be logged in at least two logs and an increasing number as the certificate lifetime goes up—five logs for certificates with lifetimes of more than 39 months. The reasons for reducing the number of logs are straightforward: increased size of the TLS handshake; increased time taken to create certificates (though note that where the number of logs exceeds the required number of SCTs, firing off requests to all logs in parallel and taking the first *n* to respond should generally be fast); and the increased size and bandwidth requirements of individual logs caused by redundant logging.

Finally, a fourth tradeoff: what should be admitted into a log? An attractive response is "anything," but logs are useful only if their size is manageable. Someone has to watch them, and if they become so large that no one can feasibly do this, then the logs may as well not exist. The straightforward answer is to admit only those certificates that can be chained to a CA that the clients recognize. There's arguably no point in logging certificates that do not satisfy this criterion, since browsers will not accept them anyway. (It would be nice if the usefulness of self-signed certificates could somehow be bolstered with this kind of log, but the spam problem, coupled with the lack of any way to effectively revoke self-signed certificates, is evidence that no one has found a way to do so; however, see the discussion on other types of logs toward the end of this article.) Besides, even if the browsers did accept them, there's currently no scalable way to revoke them.

THE ECOSYSTEM

Logs are useful only if someone is watching them, so it is important to know the participants in this ecosystem and exactly what they do.

First, certificates must somehow get into the logs. Although this will likely be done largely by CAs initially, the standard for Certificate Transparency also allows SCTs to be presented in a TLS extension. This option requires modified server software but is already experimentally supported by the Apache HTTPD server. Given software that supports the TLS extension, site operators can do their own logging, and, because they can update the SCTs whenever they feel like it—which is not possible for SCTs baked into certificates—they can also use fewer of them and run less risk that the certificate might not be accepted.

Second, clients must check that the certificates they see really are in the log. Because of the requirement that no out-of-band communications are allowed, this means trusting the log when the certificate is presented but later verifying its honesty by demanding a proof of inclusion.

Third, interested parties (site operators, CAs, and researchers, for example) need to monitor logs to ensure that CAs are not doing anything improper—issuing certificates for sites they should not be, for example, or issuing certificates that have extensions or flags set that should not be, or issuing certificates that do not conform to standards. Monitoring also ensures that logs are not violating the append-only property or breaking their MMDs, or other such misbehavior.

Finally, everyone who interacts with logs should check with each other that they have all seen the same log—that is, that the log is not presenting different views to different people. This would

enable a log to persuade clients that it had logged certificates in one view, while showing a view without those certificates to the Web sites they were purportedly for.

GOSSIP

All of these tasks are straightforward, except the last. Monitoring logs, obtaining consistency and inclusion proofs, and so forth can be done by directly querying the log, but checking for consistent views is more difficult. To do this, the various log clients will *gossip*. In the long run, this could occur over a variety of protocols—XMPP, SMTP, peer-to-peer connections, etc.—but our first suggestion is to piggyback gossip on TLS. Whenever a client connects to a server, it sends a few items to the server, which the server may verify or merely cache; in return the server sends a few items back from its cache. This establishes what is effectively a peer-to-peer network between the clients.

Because this exchange is piggybacked on a connection made for some other purpose (presumably fetching a Web page), only a few items should be sent in order to conserve bandwidth and avoid excessive latency. When the connections are specifically established for gossip (for example, directly to a log server or using some peer-to-peer protocol with other clients), there's no reason to worry about that, and clients and servers can choose to send large numbers of items—perhaps everything they know.

What items do they send? That is a matter for debate (and simulation), but we can be reasonably sure what the minimum requirement is an STH (signed tree head). Every client should to be able to reassure itself that it has seen the same logs as every other client. Logs are summarized by STHs, so gossiping them is clearly the least a client would wish to do.

STHs have some nice properties for gossiping. First, they are signed, so a bad actor cannot inject spam into the protocol. Every participant can trivially reject messages that did not originate at a log. Second, given two STHs from the same log, it is possible to prove consistency between them, and thus discard the older one. This means that caches are O(number of logs) in size.

Would more gossip be desirable? Possibly it would be useful to gossip STH consistency proofs for recent STHs, thus reducing load on logs. Servers might also want to gossip their own SCT inclusion proofs along with the corresponding STH.

Exactly what is gossiped, and when, is an open question at the time of writing. It is being explored through simulation.

OTHER USES OF TRANSPARENCY

Certificate Transparency originally motivated the work we've done on verifiable logs, but there are other useful applications:

- **Binary Transparency.** This allows you to make logs of applications that are downloadable on the Internet. Like Certificate Transparency, Binary Transparency does not prevent the binaries from being malicious, but it does reassure users that the binaries they are getting are visible to the world for analysis, making the deployment of targeted malware much harder.
- **DNSSEC Transparency**. DNSSEC is an attractive alternative to the CA-based world of authentication, but it has its own list of potential weak points—in particular, domain registries and registrars. Transparency for keys held in DNSSEC would ensure these could be monitored for correct operation.
- **Revocation Transparency**. Once a bad certificate is identified, it should be revoked. Existing

mechanisms allow dishonesty in that process, too—for example, selectively setting the revocation status to unrevoked for malicious purposes.

• **ID to key mappings**. This could consist of, for example, e-mail to PGP (Pretty Good Privacy), or instant messaging ID to OTR (off-the-record) (see https://otr.cypherpunks.ca/).

• **Trusted timestamps**. Protocols exist for digital notaries, but they currently require trust in the notary. A notary that logged all timestamps would not need to be trusted.

OTHER CONSTRUCTIONS

When thinking about Revocation Transparency, my colleague Emilia Käsper and I invented a new construct: a sparse Merkle tree (http://www.links.org/files/RevocationTransparency.pdf). The idea is that if you want to have a verifiable mapping, you can store the elements in the map as the leaves of a very large Merkle tree—say, one with 2^{256} leaves (i.e., 256 levels deep). Although such a tree would normally be impossible to calculate, the observation is that most of the leaves are empty, which means that most of the next level up would have the same value—the hash of two empty leaves; likewise for the level above that, and so on. This means it is, in fact, possible to calculate the root of the tree and make proofs of inclusion and so forth, so long as the tree is sparse. This structure can be used as an adjunct to a verifiable log to provide an efficient, verifiable map.

STATUS

Certificate Transparency is under active development at Google. We have two logs running in production, with a third planned by year's end. Others (for example, ISOC, Akamai, and various CAs) are also planning to run public logs. We have open-source implementations of all the key components. Chrome supports Certificate Transparency and will make it mandatory for EV (Extended Validation) certificates in January 2015. More than 94 percent of CAs (by volume of certificates issued) have agreed to include SCTs in their EV certificates.

Once we have seen the system working well for EV certificates, we plan to roll out Certificate Transparency for *all* certificates. We also intend to pursue some of the other uses for verifiable logs and maps.

REFERENCES

1. Comodo Group. Update 31-MAR-2011; https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html.
2. Langley, A. 2013. Enhancing digital certificate security. Google Online Security Blog; http://googleonlinesecurity.blogspot.de/2013/01/enhancing-digital-certificate-security.html.
3. Langley, A. 2013. Further improving digital certificate security. Google Online Security Blog; http://googleonlinesecurity.blogspot.co.uk/2013/12/further-improving-digital-certificate.html.
4. Laurie, B. 2011. Improving SSL certificate security. Google Online Security Blog; http://googleonlinesecurity.blogspot.co.uk/2011/04/improving-ssl-certificate-security.html.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

BEN LAURIE is a software engineer, protocol designer and cryptographer. He is a founding director of The Apache Software Foundation, a core team member of OpenSSL, a member of the Shmoo Group,

a director of the Open Rights Group, Director of Security at The Bunker Secure Hosting, Trustee and Founder-member of FreeBMD, Visiting Fellow at Cambridge University's Computer Laboratory, and a committer at FreeBSD. Laurie works for Google in London on various projects, currently focused on Certificate Transparency.