http://detector.io

kaie@kuix.de

Most Certificate Authorities (CA) have the power to issue certificates for any domain

This means, most CAs are single point of failures, they might get hacked, or their power could get abused (by creating certificates for a key that is owned by an attacker, not by the domain owner).

We must find a solution that removes the single point of failure, but is easy enough to be practical for most users of SSL/TLS.

TLS does cert verification, by finding a chain of trust to a known and trusted CA. In many applications, this is sufficient to trust a connection.

(Optionally, software might do revocation checking. But few applications do, and if they do, they might ignore failures to check.)

Suggestion: Perform additional checks on top of certificate validation, prior to trusting a TLS connection.

We want to detect a Man In The Middle (MITM).

(We assume the operator of a server doesn't cooperate with the MITM, and the MITM doesn't own the private key of the server's correct certificate. If that assumption is false, we cannot trust the server anyway. We also assume the MITM isn't able to control all connections on the internet, but only a subset of the connections.)

Suggestion: Check if our connection to the server uses the expected certificate.

How can we know what's the expected certificate for a TLS server?

We should find a universal solution to this problem, that works on the first connection, or on a new computer, and doesn't require a cache from prior connections.

We DON'T want Trust-On-First-Use (TOFU), because the MITM could already be active.

Old Idea: Notaries

Other projects have suggested to ask „Notaries“ to find out what the expected certificate for a server should be, for example, see the Perspectives or the Convergence projects.

But why should we trust Notaries? A notary is simply another concept of a trusted third party, just as CAs are.

Even if we ask multiple notaries, how can we know the notaries aren't cooperating with the MITM, or have been hacked?

If there is only a small number of global notaries used by the majority of users, those are again single point of failures.

It's better if we check on our own, instead of relying on third party opinions.

Another argument against a notary as a trust third party: If the certificate seen on our direct connection is different, who decides what's the correct one?

We don't want to prompt/ask the user for a decision. We want an automatic decision, and for that we need multiple data points.

The suggested solution is the design presented in the paper available on the http://detector.io site:

(a) Users (client software) check on their own, what the expected certificate is likely to be.

To perform the check, we connect to the destination TLS server from multiple places in the world, e.g. from multiple continents.

(b) In addition, all server administrators must regularly monitor their own site from multiple places in the world, too, to check for unexpected certificates.

(a) How can users connect to from multiple different places in the world? Most users don't have multiple VPNs or servers spread all over the world. Most users don't have a list of stable proxies they could use, and don't have the time or knowledge to continously research such a list and keep it up to date.

We need a global infrastructure that provides proxies that everyone can use, and that's easy to use.

Luckily such an infrastructure already exists: The Tor network.

Let's use the Tor (anonymity) network, provided by torproject.org and volunteers.

If you haven't heard about Tor: It's a project that attempts to provide anonymity. Volunteers run network nodes all over the world.

For our purposes (researching the expected certificate for a TLS server), we don't strictly require the anonymity property of Tor, although it's nice to have for privacy purposes.

What we need is Tor's property of forwarding local connections to different places in the world.

The DetecTor idea:

Everyone runs multiple connections to Tor.

Each of the connections to Tor is configured to select exit nodes in separate locations. For example, connection 1 could be limited to european (or democratic) countries.

We check for consistency. If connections on all (or most) routes use the same certificates as the direct connection, we allow the direct connection. If there's inconsistency, we block it.

(b) A MITM could be physically located very close to the target TLS server.

In that scenario, probing through Tor isn't sufficient, as the MITM would be able to intercept all connections from everywhere in the world.

For Detector to work, we must require that TLS server operators monitor their own server, too, using the same approach.

TLS server operators should setup a monitoring process, that regularly (e.g. every 10 minutes) checks their own server(s), by connecting to the server through the Tor network, and check for the expected certificate.

If there is a mismatch, the server operator should immediately publicly report the false certificate, which would help to get it revoked by the responsible CA and to uncover the attempted attack. The detector.io project already provides an initial (beta) tool for such monitoring.

By combining server monitoring by those who KNOW what the expected certificate is (the server owners)

and

consistency checking by client software for unexpected certificates, we should be able to detect and block connections to servers that use an unexpected certificate.

How to implement client side probing for inconsistency.

We need an implementation that is as universal as possible, it shouldn't be limited to a few applications.

The proposal is to enhance the software libraries that implement the TLS protocol, and I'd like to start with the NSS library used by Mozilla Firefox and other software (Pidgin, Chromium, etc.)

Client side implementation detail:

Application level code attempts to start an async connection attempt to a destination server. As soon as the TLS protocol library code gets the request, it can use additional threads to start the separate probing connections through the Tor network.

As long as the probing connections are running, the WOULDBLOCK state is returned to the application code, causing it to wait for the probing to be done.

The connections used for probing through the Tor network will execute the TLS handshake, in order to obtain the server certificate. Immediately aftwards, the probing connections will be terminated.

As soon as all probing connections have completed, we allow the application's primary connection and its TLS handshake to continue and obtain the server certificate from that route.

Prior to allowing anything else, the TLS protocol library code performs a comparison of the certificates seen on the different routes.

If the certificates obtained on ALL routes are identical, we can be confident to allow the connection to proceed normally.

If the certificate seen on the direct connection is identical with MOST certificates seen in the distributed probing, we might continue anyway. This could be a configuration option, based on a client's specific security requirements.

If the certificate seen on the direct route is different from the majority of those seen during the distributing probing, we MIGHT be under attack, and should block the connection.

Special scenario: Some servers use multiple different server certificates, based on the client's capabilities (supported ciphers, TLS protocol versions and extensions).

Because we suggest to integrate the probing into the connecting client, the properties of all connections will be identical, which increase the changes to see consistent certificates on all connections.

This is an advantage over separate notary servers

False negative: Some server hostnames might use many separate servers or even use content delivery networks.

If the infrastructure uses different certificates on the separate servers, we're likely to see inconsistency during probing.

It should become a best practice for operating TLS servers, that a consistent certificate (for each set of client capabilities) should be used on all servers.

In case of certificate rollovers, where a cluster of multiple servers gets updated to a new certificate, there might be a period of time when there's expected inconsistency between servers.

With the Detector approach, in the worst case there would be a temporary inability to connect to the server during the rollover deployment.

This could be optimized by caching in the client the consistency status of certificates. However, when using a cached consistency status, it should be required to have fresh revocation status information (OCSP, e.g. stapled) available.

What's next, and how you can help.

Are you a C programmer? Want to help implement the probing integration into a TLS protocol library?

I can help you get started with the NSS library used by Firefox et.al. Please talk to me.

The utility used for probing by server administrator is a C terminal application, which also needs tweaking.

Are you a python programmer? We need help with the Tor network controller that starts up the Tor connections that we need as proxies for probing.

Right now we use the Stem library and use a very simple approach to start 5 separate Tor connections. This is inefficient. It would be more efficient to have only one Tor connection, and use another existing python controller library that allows that.

Do you know how to write Nagios plugins? If we had a plugin that was compatible with the probing utility, then many administrators would be able to easily set it up and integrate it into their existing infrastructure monitoring.

Are you a packager for Linux etc. distributions? We need to get the probing utility packaged.

http://DetecTor.IO
(paper, mailing list, github)

kaie@kuix.de
(email and xmpp/jabber)

IRC: kaie