Captive portal handling for HTTPS requests

Background

Internet connections in many public places (cafes, airports, hotels, etc) make use of captive portals, which require users to login before allowing Internet access. Captive portals work either by responding to all DNS requests with their own HTTP server's address, or by hijacking HTTP requests, while allowing unfettered DNS access.

In either case, HTTPS poses a particular problem. Before a user logs on, captive portals generally either silently block all HTTPS requests, or provide their own HTTPS response.

When behind a captive portal that blocks all HTTPS requests, going to any HTTPS page, like gmail.com, will result in a timeout. A user may repeatedly try and refresh a page, or try other HTTPS pages, and this behavior will never change, making for a poor user experience.

When behind a captive portal that intercepts HTTPS requests, the portal will generally do one of the following: Return a valid SSL response with a self-signed certificate, resulting in a scary SSL warning page. Return an HTTP response, resulting in an ERR_SSL_PROTOCOL_ERROR page. Close the connection when Chrome starts SSL negotiation, resulting in the same error.

Mechanism

Captive Portal Detection

When a main frame HTTPS load is taking a while, we preemptively open a background request for **http://www.gstatic.com/generate_204**, and check the response code. We do the same when displaying SSL warnings and ERR_SSL_PROTOCOL_ERROR pages. The URL points to a service that returns HTTP 204 ("No Content") responses, and the domain does not set any cookies or save any logs. If we're not behind a captive portal, and connected to the Internet, we should get the 204 response code. If we're behind a captive portal, we should either get the logon page or a redirect to one. If we get an error or a non-HTTP response, we're either not behind a captive portal, or can't find the logon page if we are. For the purposes of captive portal discovery, we treat all HTTP status codes except 2xx, 3xx, and 511 as errors. This works regardless of whether the captive portal works by intercepting DNS or HTTP traffic.

We also preemptively run captive portal checks when displaying SSL certificate error pages, since these are often caused by captive portals as well. There is no timeout in this case.

Captive Portal Response

If we detect a captive portal logon page, we open a new tab in the topmost tabbed window, if

there is one, and navigate it to the same 204 page we requested earlier. We track all tabs navigated to this address, and if we already opened a tab in the window with this address, we don't open a new one, even if the tab has since been navigated away from that address. Once we detect we're no longer behind a captive portal, we clear this aspect of tab state, so we'll open a new tab if we end up behind a captive portal again.

Every time the login tab is navigated, we check for the captive portal. Once we detect we're connected to the Internet, we refresh all tabs that ran into HTTPS timeouts or certain SSL certificate errors on their last navigation, except those that were POSTs. If a loading HTTPS tab has yet to reach a timeout or SSL error, we will not reload it until after it errors out. While sending POST request wouldn't result in double POSTs, since this will only trigger on errors that indicate the HTTP request was not even sent, we do not resubmit POSTS, to be on the safe side.

We do this all on a per-profile basis, since a profile may contain state or extensions needed to login to server, and bypass (Or reach) a captive portal.

Server Load

To keep the number of requests down, we throttle the frequency of checks triggered by new SSL timeouts to one every 10 minutes when we're not behind a captive portal. When we are behind a captive portal, we'll check at most once every 15 seconds. If 8 checks in a row return the same result, we start doubling the delay for every check, until the delay reaches two hours These numbers are subject to change once we have some user metrics.

Since we're only triggering on SSL timeouts, and throttle requests as described above, we expect server load to be quite modest. We plan to collect metrics from Canary and dev channel to get load estimates to verify this.

Errors on redirects

Generally when we get a redirect to an HTTPS page, it means that there is an Internet connection, so no captive portal check is needed. However, HSTS (HTTP Strict Transport Security), which can force a domain to use HTTPS rather than HTTP, is implemented as an automatic HTTP to HTTPS redirect. This is generally what happens if, for example, one types "www.gmail.com" into Chrome. Since the redirect occurs without any over-the-wire transfer, it will bypass any captive portal.

To detect captive portals in this case, we allow the first HTTPS URL of any redirect chain to trigger a captive portal request. Once we've successfully received one HTTPS response in a redirect chain, no request in the redirect chain will result in a captive portal check.

Privacy

We check for captive portals only if "Use a web service to help resolve navigation errors" is enabled. The domain we use to check for captive portals will be a cookieless domain,

and we won't send cookies on the probe requests, though it is possible for a captive portal to set cookies when we navigate to its login page.

Since we do everything per profile, no record of checks initiated by Incognito tabs will remain, once all Incognito windows are closed.

Broken login pages

It's possible that we'll get a series of redirects that end with an error of some sort, most likely an SSL certificate error or a timeout. Currently, we do not open a new tab in tis case. In the future, we may want to handle certificate errors and network errors differently.

Metrics

We record the number of times we check for captive portals and how often they fail and succeed. We also record how many times we get each result in successive checks, and the time between first detecting a captive portal and the captive portal no longer being detected.

Native Captive Portal Detection

Some platforms, such as Lion and Windows 8, have their own built-in captive portal detection and response schemes. We disable captive portal detection on these platforms. Earlier versions of Windows also have captive portal detection, but it's less reliable, so we do have captive portal detection enabled on earlier Windows versions.

Other Concerns / Future Work

False Positives

We only consider ourselves behind a captive portal when some server returns an unexpected non-error HTTP response (2xx, 3xx, 511, other than 204). If the connection fails or times out, then we are not behind a captive portal, or at least cannot find a login page, so we do nothing.

It's still theoretically possible for us to get a "you can't get there from here" page on LANs or networks with restricted Internet access. We'll have to rely on reports from end users to learn if this is a problem.

Unexpected Login / Network Changes

It's possible that the user could have logged in to the captive portal in the time between the original HTTPS load start and the time we start our captive portal test. In this case, we currently won't refresh the HTTPS page, since we never detect a captive portal.

It's also possible for the user to change networks or log in between the time we detect a

captive portal and when we open the login page. We don't plan to worry about this case, initially, but if it turns out to be an issue, we could use a hidden tab for the navigation to the login tab, and then only show it if we receive a non-204 HTTP response.

Returning to the Captive Portal Test URL

Some captive portals may return the user to the original URL they were requesting when the captive portal intercepted their request.

We currently do nothing special in this case. In this case, the captive portal would try to navigate to the original URL, get a 204 response, and the navigation would be canceled.

In the future, we may close the tab, if it's in a tabbed window with at least one other tab.

Pre-existing Login Tabs

There may be another login tab already open. Since we follow redirects when probing for the captive portal, in most cases we can probably identify such tabs, and instead of opening a new login tab, we could just activate one. Currently we don't do this, but may implement it at a later point in time.

More general use

There are a lot of other cases where captive portal detection may be useful as well. Due to concerns about latency, accuracy, and bandwidth, we're currently only focusing on a very limited set of cases - SSL timeouts and certain errors (and soon the ChromeOS login screen). We may look into using this mechanism in more general use cases in the future.

The host name used for the checks makes it look like the captive portal page is from Google

There are a couple ways around this. The UI leads have suggested obscuring the URL in the omnibox. Nothing is currently implemented to do this.

Quick Implementation Overview

Captive portal detection is implemented as ProfileKeyedService, CaptivePortalService, which broadcasts the results of its queries to all its observers. At some point, this service will also be used by ChromeOS to detect captive portals at the logon screen.

The initial check will be done by a URLFetcher, with load flags set so as to avoid sending/setting cookies, or writing to the cache.

Each tab has an observer that tracks whether it's possibly been broken by a captive portal or is a logon page. The first is determined solely by the result of the last navigation, and whether the page is an SSL page, and the latter is set by CaptivePortalService after opening the tab. Once we detect there is no longer any captive portal, the observers refresh broken pages and clear their state.

This lives in chrome/ rather than net/ because it's a per-profile service for reasons already mentioned (Needs access to any existing authentication/proxy information, keeps incognito mode separate), and because the response to captive portal detection is solely handled by chrome/ objects living on the UI thread.