[<prev] [next>] [<thread-prev] [thread-next>] [day] [month] [year] [list]

Date: Tue, 29 Nov 2022 16:38:03 +0100
From: Solar Designer <solar@...nwall.com>
To: Julien Pivotto <roidelapluie@...metheus.io>
Cc: oss-security@...ts.openwall.com
Subject: Re: CVE-2022-46146 in Prometheus' exporter toolkit: bypass basic authentication

On Tue, Nov 29, 2022 at 01:22:58PM +0100, Julien Pivotto wrote:

> The exporter toolkit is a go library intended at Prometheus exporters. It provides some features that are useful for Prometheus exporters, which work by exposing HTTP servers to be exposed by the Prometheus server.
>
> One of those features is basic authentication. To achieve this, Prometheus requires you to store a bcrypt hash into a file, web.yml.
>
> While bcrypt is fine, it takes by design a lot of time and resources to compare a password with a hash. To limit this impact, we have a built-in cache that caches the good and bad answers.
>
> Once a request comes, we check it against the cache and decide whether to allow the request. We also check that the user is valid. However, the key for that cache is predictable:
>
> hex(username + hashed password + input password)
>
> If you know the bcrypted password, you can poison the cache and use that cached positive value in a subsequent query:
>
> Request 1:
>
> username = username+hashed password
> password = "fakepassword"
>
> Request 2:
>
> username = username
> password = bcrypt(fakepassword)+"fakepassword"
>
> "fakepassword" is used as bcrypted password when a user does not exist.
>
> The fact that we save unhappy tentatives and that we validate non-existing users against "fakepassword" is to prevent side channel attacks that could reveal if a user exists in a system or not.
>
> Prometheus 2.37.4 and 2.40.4 are out, with this fix. We recommend all the exporters that depend on the repository to upgrade.
>
> CVE-2022-46146 was assigned to this security report in our exporter toolkit:
> https://github.com/prometheus/exporter-toolkit/security/advisories/GHSA-7rg2-cxvp-9p7p
>
> We would like to thank Lei Wan for the responsible disclosure of this bug.

The above describes the issue, but not the fix. This left me curious.

The fix commit appears to be this:

https://github.com/prometheus/exporter-toolkit/commit/5b1eab34484ddd353986bce736cd119d863e4ff5

It makes two changes:

1. Rather than concatenate the original strings to produce the cache key, the 3 individual components are first hex-encoded and are then concatenated with colons as separators.

2. Cache records for authentication against non-existent users with "fakepassword" no longer indicate that authentication passed.

This appears sufficient to address the described issue.

The caching is controversial. A comment in cache.go says:

```
// newCache returns a cache that contains a mapping of plaintext passwords
// to their hashes (with random eviction). This can greatly improve the
// performance of traffic-heavy servers that use secure password hashing
// algorithms, with the downside that plaintext passwords will be stored in
// memory for a longer time (this should not be a problem as long as your
// machine is not compromised, at which point all bets are off, since basicauth
// necessitates plaintext passwords being received over the wire anyway).
```

IMO, storage of plaintext passwords in memory for longer doesn't become a non-issue just because plaintext passwords are also available during authentication. A compromise might be short-lived and not every user who had logged in before would necessarily log in again while the server is compromised, so storage of plaintext passwords that might have been used a long time ago does make things worse and partially defeats the purpose of password hashing. OTOH, in a persistent Go service they're likely to stay around anyway.

Another (minor) concern is that the cache is indexed by password-derived material, making the password a bit more susceptible to local CPU cache timing attacks (after a leak of bcrypt's random salts to the attacker). bcrypt itself also does such indexing, but that's part of why it turned out to be relatively inefficient on GPUs, so it can be viewed as justified risk. Is the risk from the cache also justified, by it saving computing resources (not under deliberate DoS, though)? Maybe. Alternative designs of the cache are possible, but involve other non-trivial trade-offs and subtle detail. If Go uses keyed hashing for its maps (does it? I don't know), that also provides a mitigation (while the random key is not leaked/inferred).

Further, perhaps Go maps internally compare the provided key against some stored keys (within one hash bucket?), and that can probably involve a byte-by-byte comparison, kind of leaking the length of matched substring even to the remote client and potentially allowing to probe candidate passwords character-by-character (when hitting the same hash bucket). This is mitigated by such probing also thrashing the cache and triggering the much slower bcrypt computation.

To clarify, I am not suggesting that any changes to the code be made - they could as well introduce new issues, and would need a new review. Dropping of the cache is a pretty obvious change to make, but maybe the risks involved are no big deal for this specific service.

I was just curious and I thought some others in here would be as well, so I am sharing observations and thoughts.

Alexander