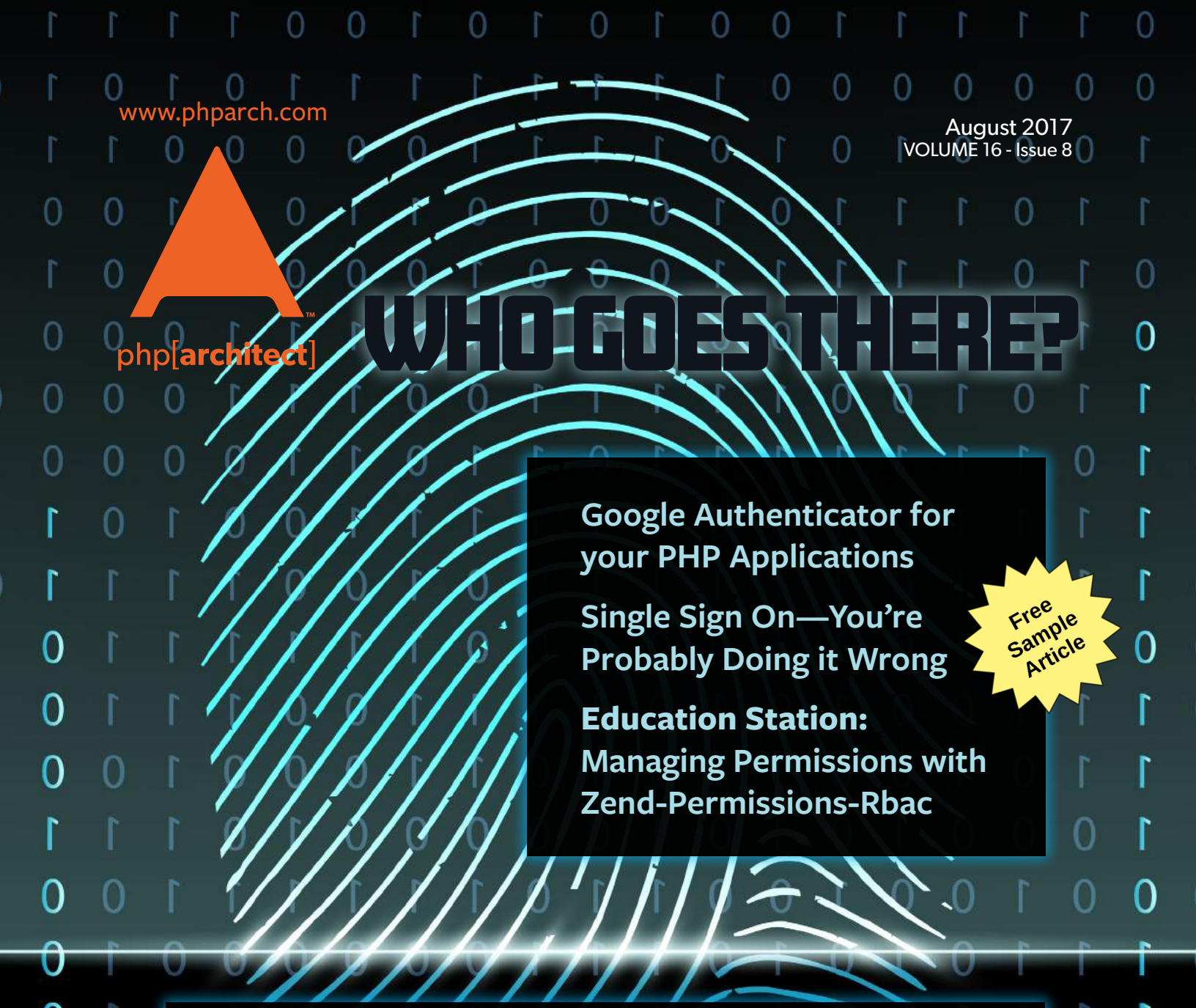
www.phparch.com

August 2017
VOLUME 16 - Issue 8

php[architect]

WHO GOES THERE?



Google Authenticator for your PHP Applications

Single Sign On—You're Probably Doing it Wrong

Education Station: Managing Permissions with Zend-Permissions-Rbac

Free Sample Article

ALSO INSIDE

PHP[WORLD] 2017

This is a conference like no other. Designed to bring together all the communities linked by the PHP programming language.

November 15-16, 2017
Washington, D.C.

world.phparch.com

Sponsored by:

Magento®

MySQL™

php[architect]

CONTENTS

AUGUST 2017
Volume 16 - Issue 8

WHO GOES THERE?

Features

Columns

Editor-in-Chief: Oscar Merida
Editor: Kara Ferguson

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Managing Partners
Kevin Bruce, Oscar Merida, Sandy Smith

php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Single Sign On—You're Probably Doing it Wrong

Arne Blankerts

Requiring users to log in individually to all the websites they need for their work is more than merely annoying: It wastes a lot of time and turns maintaining log-in credentials and permissions into a nightmare for the administrative staff. Let's see if we can fix that with a single sign-on service.

Setting up multiple services to share a common login without re-authenticating for every request or when switching to a different application may seem like a rather complex task at first glance. And while of course we have some things to consider, the final solution does not have to be overly complex.

The basic concept of a single sign-on service—SSO for short—is to allow a user to log in (authenticate) once upon requesting the first protected URL, independent of the actual service visited, and maintain this state independent of whatever other service is being used next. When set up as a proxy, it's the SSO's job to ensure the user is authenticated before allowing the request to be passed on to the actual service, and to provide the application with identity information so it has a means of knowing who its user is.

An SSO implies that the management of user accounts is centralized. This has quite some benefits for administrative staff, as they only have to create one user account for all applications. It also avoids the need to synchronize the login details among various platforms and taking their varying rules for passwords, user names, and other required information into account.

Centralized authentication also benefits the end user: No more password changes for individual applications with different requirements, and gone is the need to remember where the password was already changed and where the old one is still valid. Because, let's face it—not everybody adheres to the rule to not use the same password for multiple services, anyway. And of course, there's no need to log in over and over again, just to open all the applications needed for the work day.

On the other hand, relying on a single service turns authentication into a so-called single point of failure—SPoF for short. If the SSO service is not available, no login can be performed for any application. For us and from a system architectural point of view, this means the availability of the SSO server must be our highest priority when planning day-to-day operation.

This seems far from being a new problem, though, and chances are, you even have or had other single point of failures already in your company or network: a shared database server, for instance, or the single uplink to the internet from your office, that one network switch or router, or even the single file server where all the documents are stored.

Luckily, many possible solutions to achieve high availability already exist: From simple redundancies to fail-overs and clusters, all we need to decide is which one would fit our concrete needs best.

As the benefits clearly outweigh the potential problems, it's time to finally dive into developing our single sign-on solution.

Authentication, Authorization, or Both?

Granted, the common abbreviation "Auth" isn't very specific. Does it stand for Authentication, Authorization, or even both? And which one do we need in our context SSO? To make things complicated, the surprising answer is: "It depends."

Authentication, the act of verifying that a person is who she or he claims to be, is usually a prerequisite to gaining access. Such a process can be implemented in a fairly generic fashion and is well understood. Determining whether or not that person, after being authenticated, is now authorized to perform a certain action or is allowed to retrieve the requested data is a more complex task.

While basic ACL (access control lists) constructs can be potentially managed in a generic fashion, more fine-grained restrictions may also be required. For example, access to a profile was granted by the ACL, but the amount of data shown varies upon the type of user. Particularly in CRUD-based applications, determining upon save which part of the profile changed and whether or not the current user was actually entitled to change that field can be quite hard as well. Trying to find a generic solution to both managing and mapping these types of permissions back into the application is likely to cause sleepless nights for everybody involved.

So to not drown in complexity, the SSO should be limited to offering authentication. And maybe check whether or not the user is allowed to use the application in question to catch policy violations early on. Any additional authorization has to be delegated to the application itself.

Before developing new software, it's

always a good idea to look at preexisting solutions. Maybe we don't have to program anything?

OAUTH?

Of course there are various technical approaches to SSO, as well as many commercial solutions. When tasked to implement a "simple" SSO service themselves, many developers consider OAUTH the way to go. OAUTH[1] aims to be an "open protocol to allow secure authorization in a simple and standard method from web, mobile, and desktop applications". As the mission statement clearly focuses on authorization, this is likely not what we need, but let's have a look anyway: OAUTH has been designed to enable users to grant access to functionality of the providing application to a third party without revealing the user's credentials. A very important aspect in the design of OAUTH is that the third party software is not to be trusted, from the providing application's perspective.

Beyond the authentication itself, though, we don't want to access any functionality of the SSO server, let alone want the user to be in control of it. As this goes conceptually in the opposite direction from where we are trying to go, OAUTH cannot be the answer. That saves us from looking into its technical aspects and we can check the next candidate.

SAML?

Next on the list of things Google usually returns when we're searching for SSO and PHP is SAML—the Security Assertion Markup Language. SAML has been developed by the OASIS consortium (OASIS stands for "Organization for the Advancement of Structured Information Standards"). OASIS has published a broad range of standards, like the Open Document Format used by LibreOffice or the Docbook format used by the PHP Manual. SAML has been around for a while, since 2001, and is used by many big players in the enterprise market. It is a standard for exchanging authentication and authorization data between security domains. The powerful XML-based protocol uses so-called security tokens containing assertions to pass information about an end user between an identity provider and a service.

It not only sounds rather complex, it comes with a lot of technical overhead if all we want is a shared login.

Shouldn't that be rather simple?

Naive Approach

Let's take a step back and re-evaluate our problem. Technically speaking, our goal is to have multiple applications know the user is logged in and authenticated. We do not care about authorization at this point.

Assuming all the applications are written in PHP and are hosted within the same network or even the same physical machine, the most basic and simple solution could be to use a shared session.

Except, this does not work: As soon as there are additional application-specific values and potentially serialized objects stored within the PHP session, the idea collapses. Since by definition all data in the session is shared, every application needs to able to read and work with the given data structure. We would be restricted to scalar values if we did not want incomplete class objects to be created upon unserialization.

But that's actually the least of our problems. If we were to implement this, we'd have a serious security issue on our hands: We provided an application with the means of modifying the session state of an unrelated application! That's a big problem and definitely a no-go. To fix that, we would need to separate the authentication information from the application session. That means we'd have to deal with two sessions in one PHP process. Sadly, the standard PHP session extension does not support that and we'd have to implement all the session handling ourselves.

It might be a fun task to do but hardly qualifies as a simple solution. And sharing a session has another potential drawback: What happens if not all applications are written in PHP?

Shared sessions are out.

JSON Web Token

All server-centric approaches failed for varied reasons, maybe we should include the browser into the game and use its ability to store information for us. Given we cannot simply trust anything stored on the client as plain text because it could be manipulated, we'd need to find a secure way to store our authentication result in the browser.

One such way would be a JSON Web Token, or JWT for short. Defined in RFC 7519, JWTs are an open standard that defines a compact and self-contained way of transmitting information between two parties in the form of a JSON object. The information provided can be verified by checking a digital signature and should thus be safe from manipulation. JWTs can be signed by either using a secret within the HMAC algorithm or a public/private key pair. All the nice encryption is actually rather useless if an attacker can simply capture the network traffic and extract the token because of an unprotected transport. For this approach to work, we have to rely on HTTPS all the way. Luckily, that shouldn't be too much of a problem, as plain HTTP is dying out anyway.

The web page about JWT[2] explicitly mentions authentication as one common usage scenario, along with single sign on. There are client implementations for pretty much every language; PHP has even various alternative solutions to choose from.

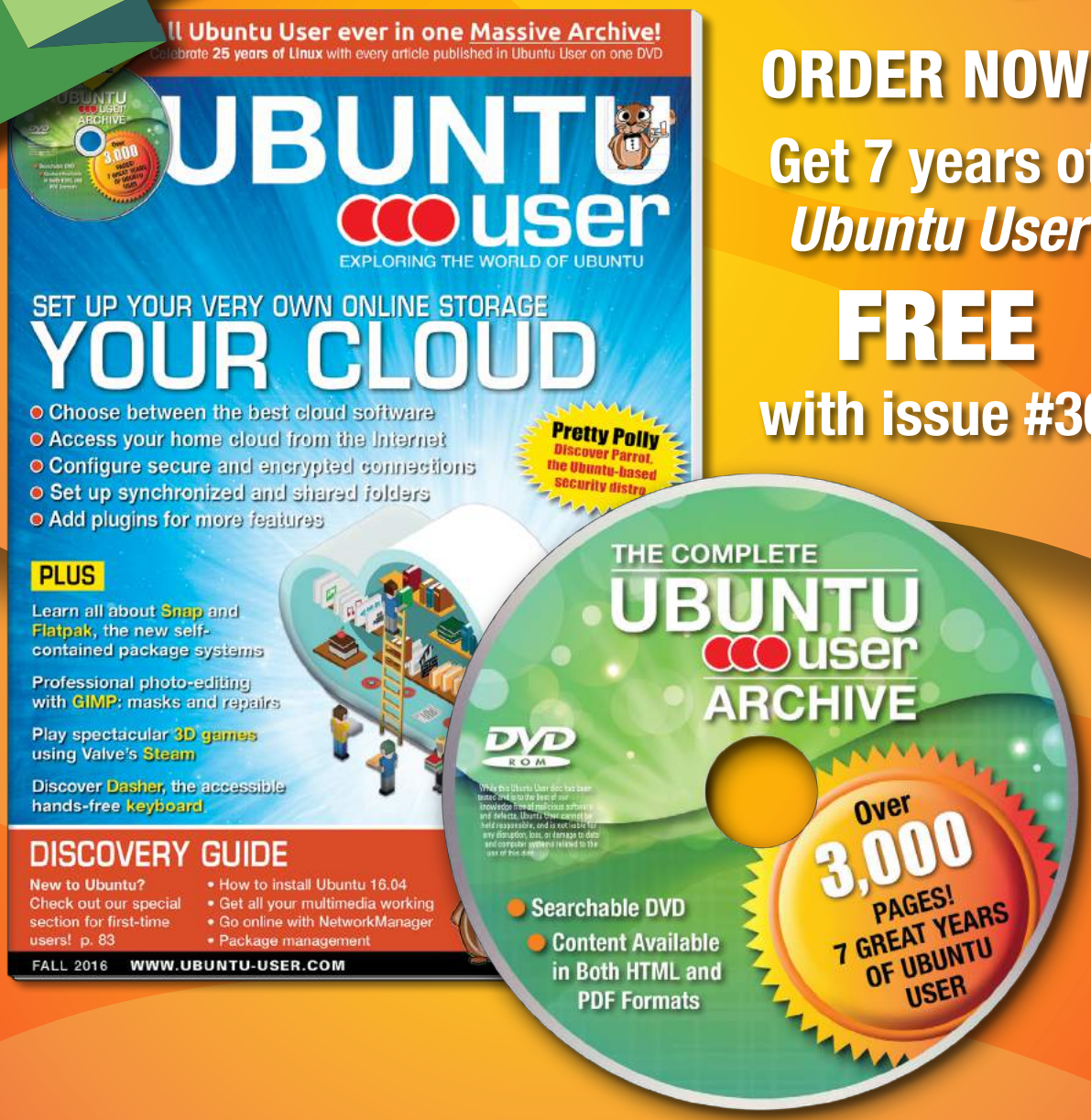
Sounds good! So maybe that's the way to go?

Getting all information about the user without the need to call an API or query an additional data source certainly sounds like a plus. But how do we get the information from the browser to the server? While JWT defines the required properties in the JSON string, there is no official standard for sending it. A recommended way seems to be the

1 *OAUTH: http://oauth.net*

2 *JWT: http://jwt.io*

Celebrating 25 Years of Linux!



ORDER NOW!

Get 7 years of *Ubuntu User*

FREE

with issue #30

***Ubuntu User* is the only magazine for the Ubuntu Linux Community!**

BEST VALUE: Become a subscriber and save 35% off the cover price!
The archive DVD will be included with the Fall Issue, so you must act now!

Order Now! Shop.linuxnewmedia.com

use of a custom authorization header:

```
Authorization: Bearer <token>
```

While such a header can easily be added to any JavaScript-borne request or any API call via http, there seems to be no way to make the browser add such a header to the outgoing request by itself. And why would there be, given that that is what cookies were originally designed for. And, sure enough, the JWT manual also mentions cookies as a valid option. Cookies, though, are limited in their size which might prove to be an issue, depending on how detailed the user information is supposed to be. On the other hand, if the data size were to outgrow the size restrictions of cookies, the traffic overhead for every request would be of considerable magnitude.

And that's not all: Conceptually, JWTs are like a distributed cache. When using JWT for storing authentication data on the client, the SSO server is never again bothered – that is, until the JWT expires. That's of course by intention and makes a lot of sense for JWTs. But in case you want to change access permissions, disable an account ,or merely add new values, you'll have to wait until the token has expired.

This, along with all the parsing and signing overhead, seems again like the opposite of a simple solution.

User Tokens and Callbacks

That leaves us with the idea of a user token or identifier and an API call from the application to our SSO service to get the actual user information. The process could be straightforward: Once the user has been authenticated, a user token or identifier is generated by the SSO service and stored as a regular, shared cookie on the client. Equipped with this token, the application could call an API provided by our SSO service to retrieve additional information about the current user.

While the general idea of a simple session-like token to be stored on the client sounds good, the rest seems awfully like reinventing OAUTH. It also means we have to provide this additional API and define the data structures, satisfying the needs of all the applications we plan on protecting. And we have to make every application call our API to get the information.

Listing 1

```
server {

    # ...

    server_name application.example.com;

    location / {
        lua_code_cache off;
        access_by_lua_file /var/www/lua/access.lua;

        proxy_set_header X-Real-IP  $remote_addr;
        proxy_pass https://application.local/;
    }
}
```

So, maybe requiring callbacks is not yet a good solution.

Avoiding Callbacks

Truly separating concerns, the protected application should be unaware of the SSO proxy setup. To achieve that, the SSO would have its own session token, as described in the last paragraph, but use a proxy approach to inject additional information for the application into the request before forwarding it.

That way, the application will get what it needs without making any API calls itself and the SSO proxy would be transparently in charge of security at all times. Sounds good, but also rather complicated? Actually, it's not, as the following examples will (finally) demonstrate.

Let's start with the web server for the application. Since we need an http server that can handle dynamic routing and proxying, we'll be using NGINX, Redis, and some simple LUA:

The public-facing web server—reachable at "application.example.com"—will execute the lua script `access.lua` to determine whether or not the request can and should be forwarded to the actual application server—named `application.local` in this example. It should be obvious that access to that server should be restricted to only work via the SSO proxy in front of it.

The main SSO logic is implemented in the `access.lua` script in Listing 2.

The code above checks to see whether a UUID cookie is set and, if so, whether matching data can be found in the Redis key value store. If that did not work or if no such cookie was

Listing 2

```
local uuid = ngx.var.cookie_uuid;
local sso = 'https://sso.example.com/';

if not uuid then
     ngx.redirect(sso);
end

local redis = require "resty.redis"
local red = redis:new()
red:set_timeout(1000) -- 1 sec
local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end

local res, err = red:get(uuid)
if not res then
    ngx.redirect(sso);
end

if res == ngx.null then
    ngx.redirect(sso);
end

ngx.header["uuid"] = nil
ngx.req.set_header("X-USER-INFO", res);
```

Listing 3

```php
<?php

// ...

if ($this->credentialsAreValid()) {
    $uuid = trim(
        file_get_contents('/proc/sys/kernel/random/uuid')
    );
    $data = $this->getUserData();
    $ttl  = 3600;

    $redis = new Redis();
    $redis->connect('127.0.0.1', 6379);

    $redis->set($uuid, $data);
    setcookie("uuid", $uuid,
        time() + $ttl, '/', 'example.com', true, true);

    // ...
}

// ...
```

Listing 4

```
server {

    // ...

    ssl_client_certificate /etc/ssl/ca/certs/ca.crt;
    ssl_crl /etc/ssl/ca/private/ca.crl;
    ssl_verify_client on;

    error_document 403 = @register;

    location / {
        fastcgi_param  VERIFIED $ssl_client_verify;
        fastcgi_param  DN $ssl_client_s_dn;

        // ...
    }

    location @register {
        return 302 https://register.example.com;
    }

}
```

set, the user will be redirected to the SSO service available at "sso.example.com."

In cases where the cookie was set and there is matching data found in Redis, it will be injected into the request as an additional, custom header named `X-USER-INFO`. And the request will be returned to Nginx, which in turn will proxy forward it to our application at 'application.local'.

Our `access.lua` above relies on the UUID cookie and on Redis to hold the user details. For that to work, the user data needs to be stored in Redis and the cookie needs to be set upon successful login. The following excerpt (Listing 3) from the login demonstrates how this would roughly look like in source code.

To complete the setup, we of course still need the two remaining hosts, "sso.example.com" as well as "application.local", to be configured. Both are rather standard PHP-enabled web server setups and thus do not provide anything of real interest at this point. Examples can be found in the accompanying material to this article, for those who are interested never the less.

Assuming we implement an actual authentication mechanism at sso.example.com, we now have a working single sign-on system for all applications within our domain.

Sending the user details as a header is of course only one way of providing the data. You are completely free to implement whatever format suits your needs. For instance, we could enhance the lua code to create a JSON web token to be forwarded to the internal application upon login, satisfy some BASIC HTTP Auth , or emulate an LDAP or even TLS Certificate DN.

TLS Client Side Certificates

Speaking of certificates: in case you want to take single sign-on one step further, you might consider avoiding the actual log-in process completely by deploying client side certificates.

Unknown to many, TLS certificates can work both ways: the commonly used way a server authenticates itself to the browser but also the other way around, having a client authenticate itself to the server.

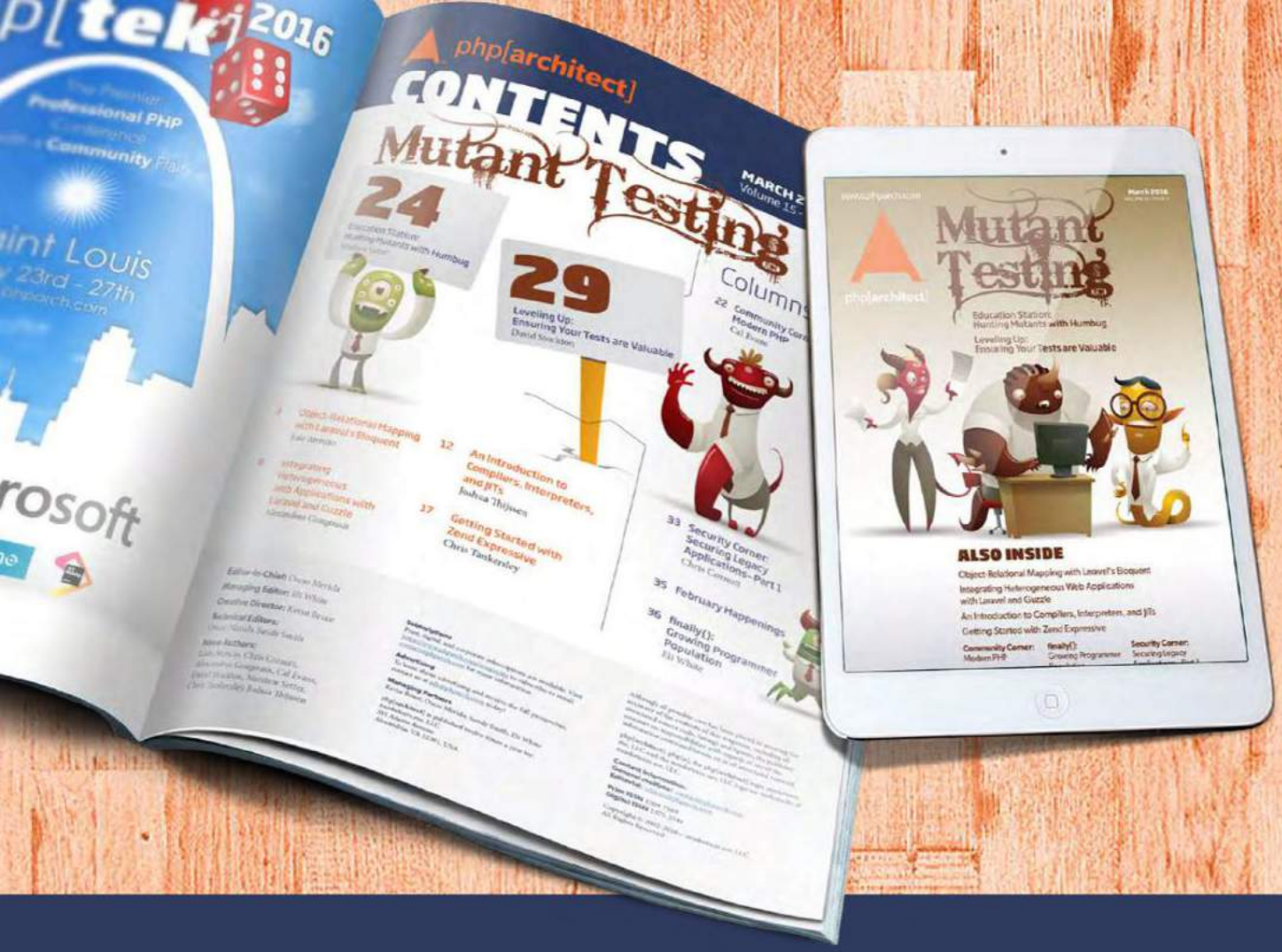
For this process to work, you need to set up your own certificate authority (CA) and provide the user with a certificate signed by that authority. You could also create a registration portal and use the HTML5 form element 'keygen' to have the browser create a key and then sign the generated CSR. However, the development of such a portal is out of the scope of this article.

If you have your CA-signed client certificate installed in the browser, you can require its use by enabling the respective functionality within NGINX with only a few changes to any TLS-enabled server configuration. See Listing 4.

With the above server setup, every request must have a valid client-side certificate installed that is signed by the referenced CA. If that is not the case, the user will be redirected to the registration portal.

Client side certificates do not come with the domain name restriction as they do not use cookies. But they are bound to the browser or even the OS, making it a lot harder to share the device if it does not come with a multi-user environment or handling individual profiles.

Arne Blankerts has already dealt with computers when networking was still an adventure. As Co-Founder and Principal Consultant of The PHP Consulting Company (thePHP.cc), Arne helps his clients to develop software successfully. He is author and maintainer of various Open Source development tools, and is a regular presenter at conferences. @arneblankerts



Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[**architect**]

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe