Klik hier voor pdf...

SECURITY

Les 9 Webdevelopment

WAAR STAAN WE?

1. Frontend: HTML, CSS, Javascript
2. Frontend: Bootstrap
3. Backend: C#
4. MVC
5. Backend: C# LINQ
6. ORM
7. Testen
8. Layout
9. **Zoeken/filteren, Sorteren, Pagineren**
10. Web API, JSON, Ajax, Azure
11. Security
12. Architectuur

VANDAAG OP HET PROGRAMMA

- Overige security
- 🛡️ Authenticatie en Authorisatie
- OWASP Top 10
- Opdracht

OVERIGE SECURITY

SIE

Wat weten we al?

SECURITY BY DESIGN PRINCIPES (OWASP)

- **Minimise attack surface area:** restricties aanbrengen in (online)functies
- **Establish secure defaults**: de defaults zijn zo veilig als mogelijk
- **The principle of least privilege**: de lijst met rechten is zo klein als mogelijk maken (access)
- **The principle of defence in depth**: verschillende type security controles (lagen) aanbrengen
- **Fail securely**: zorg dat eventuele fouten niet tot onveilige situaties leiden (o.a. goede detectie / logging)
- **Don’t trust services**: ga er in principe van uit dat third-party (externe) services niet te vertrouwen zijn
- **Separation of duties**: breng een taakscheiding aan (fraude voorkomen)
- **Avoid security by obscurity**: geen zwakke plekken in de beveiliging ‘verstoppen’
- **Keep security simple**: complexe architecturen vergroten de kans op fouten
- **Fix security issues correctly**: security issues (root cause) moeten grondig worden gerepareerd en getest.

SECURITY BY DESIGN PRINCIPES (OWASP)

- **Minimise attack surface area:** restricties aanbrengen in (online)functies
- **Establish secure defaults**: de defaults zijn zo veilig als mogelijk
- **The principle of least privilege**: de lijst met rechten is zo klein als mogelijk maken (access)
- **The principle of defence in depth**: verschillende type security controles (lagen) aanbrengen
- **Fail securely**: zorg dat eventuele fouten niet tot onveilige situaties leiden (o.a. goede detectie / logging)
- **Don't trust services**: ga er in principe van uit dat third-party (externe) services niet te vertrouwen zijn
- **Separation of duties**: breng een taakscheiding aan (fraude voorkomen)
- **Avoid security by obscurity**: geen zwakke plekken in de beveiliging 'verstoppen'
- **Keep security simple**: complexe architecturen vergroten de kans op fouten
- **Fix security issues correctly**: security issues (root cause) moeten grondig worden gerepareerd en getest.

'HACKEN' VAN URL'S

- Het principe is dat een hacker een URL intypt in de adresbalk van de browser
- Of dat hij/zij een tool als Postman gebruikt
- De URL heb je zodanig aangepast dat je meer mag dan de bedoeling is
- http://mijnsite.nl/AB6G4JDLHGE7759HMDWSHGWLS/allesverwijderen
- http://mijnsite.nl/klant/edit?admin=true
- http://mijnsite.nl/miljoenennota/2017/document.pdf
- Echt gebeurd: de miljoenennota was op die manier al een paar dagen eerder beschikbaar dan officieel de bedoeling was. Het bestand stond er al maar er was nog geen link gepubliceerd. NOS
- Les: geheime URL's bestaan niet! Dwing eerst login af.

'HACKEN' VAN URL'S

- Het principe is dat een hacker een URL intypt in de adresbalk van de browser
- Of dat hij/zij een tool als Postman gebruikt
- De URL heb je zodanig aangepast dat je meer mag dan de bedoeling is
- http://mijnsite.nl/AB6G4JDLHGE7759HMDWSHGWLS/allesverwijderen
- http://mijnsite.nl/klant/edit?admin=true
- http://mijnsite.nl/miljoenennota/2017/document.pdf
- Echt gebeurd: de miljoenennota was op die manier al een paar dagen eerder beschikbaar dan officieel de bedoeling was. Het bestand stond er al maar er was nog geen link gepubliceerd. NOS
- Les: geheime URL's bestaan niet! Dwing eerst login af.

'HACKEN' VAN URL'S

- Het principe is dat een hacker een URL intypt in de adresbalk van de browser
- Of dat hij/zij een tool als Postman gebruikt
- De URL heb je zodanig aangepast dat je meer mag dan de bedoeling is
- http://mijnsite.nl/AB6G4JDLHGE7759HMDWSHGWLS/allesverwijderen
- http://mijnsite.nl/klant/edit?admin=true
- http://mijnsite.nl/miljoenennota/2017/document.pdf
- Echt gebeurd: de miljoenennota was op die manier al een paar dagen eerder beschikbaar dan officieel de bedoeling was. Het bestand stond er al maar er was nog geen link gepubliceerd. NOS
- Les: geheime URL's bestaan niet! Dwing eerst login af.

'HACKEN' VAN URL'S

- Het principe is dat een hacker een URL intypt in de adresbalk van de browser
- Of dat hij/zij een tool als Postman gebruikt
- De URL heb je zodanig aangepast dat je meer mag dan de bedoeling is
- http://mijnsite.nl/AB6G4JDLHGE7759HMDWSHGWLS/allesverwijderen
- http://mijnsite.nl/klant/edit?admin=true
- http://mijnsite.nl/miljoenennota/2017/document.pdf
- Echt gebeurd: de miljoenennota was op die manier al een paar dagen eerder beschikbaar dan officieel de bedoeling was. Het bestand stond er al maar er was nog geen link gepubliceerd. NOS
- Les: geheime URL's bestaan niet! Dwing eerst login af.

≡

'HACKEN' VAN URL'S

- Het principe is dat een hacker een URL intypt in de adresbalk van de browser
- Of dat hij/zij een tool als Postman gebruikt
- De URL heb je zodanig aangepast dat je meer mag dan de bedoeling is
- http://mijnsite.nl/AB6G4JDLHGE7759HMDWSHGWLS/allesverwijderen
- http://mijnsite.nl/klant/edit?admin=true
- http://mijnsite.nl/miljoenennota/2017/document.pdf
- Echt gebeurd: de miljoenennota was op die manier al een paar dagen eerder beschikbaar dan officieel de bedoeling was. Het bestand stond er al maar er was nog geen link gepubliceerd. NOS
- Les: geheime URL's bestaan niet! Dwing eerst login af.

'HACKEN' VAN URL'S

- Het principe is dat een hacker een URL intypt in de adresbalk van de browser
- Of dat hij/zij een tool als Postman gebruikt
- De URL heb je zodanig aangepast dat je meer mag dan de bedoeling is
- http://mijnsite.nl/AB6G4JDLHGE7759HMDWSHGWLS/allesverwijderen
- http://mijnsite.nl/klant/edit?admin=true
- http://mijnsite.nl/miljoenennota/2017/document.pdf
- Echt gebeurd: de miljoenennota was op die manier al een paar dagen eerder beschikbaar dan officieel de bedoeling was. Het bestand stond er al maar er was nog geen link gepubliceerd. NOS
- Les: geheime URL's bestaan niet! Dwing eerst login af.

'HACKEN' VAN URL'S

- Het principe is dat een hacker een URL intypt in de adresbalk van de browser
- Of dat hij/zij een tool als Postman gebruikt
- De URL heb je zodanig aangepast dat je meer mag dan de bedoeling is
- http://mijnsite.nl/AB6G4JDLHGE7759HMDWSHGWLS/allesverwijderen
- http://mijnsite.nl/klant/edit?admin=true
- http://mijnsite.nl/miljoenennota/2017/document.pdf
- Echt gebeurd: de miljoenennota was op die manier al een paar dagen eerder beschikbaar dan officieel de bedoeling was. Het bestand stond er al maar er was nog geen link gepubliceerd. NOS
- Les: geheime URL's bestaan niet! Dwing eerst login af.

≡

'HACKEN' VAN URL'S

- Het principe is dat een hacker een URL intypt in de adresbalk van de browser
- Of dat hij/zij een tool als Postman gebruikt
- De URL heb je zodanig aangepast dat je meer mag dan de bedoeling is
- http://mijnsite.nl/AB6G4JDLHGE7759HMDWSHGWLS/allesverwijderen
- http://mijnsite.nl/klant/edit?admin=true
- http://mijnsite.nl/miljoenennota/2017/document.pdf
- Echt gebeurd: de miljoenennota was op die manier al een paar dagen eerder beschikbaar dan officieel de bedoeling was. Het bestand stond er al maar er was nog geen link gepubliceerd. NOS
- Les: geheime URL's bestaan niet! Dwing eerst login af.

⚔ SQL INJECTIE

- Is onmogelijk in onze ORM (EF Core) zolang we niet expliciet SQL schrijven
- Is mogelijk met `context.Database.ExecuteSqlCommand`:
  - ✗ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = " + naam)`
  - ✗ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = {0}", naam)`
  - ☑ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = @x", new SqlParameter("@x", naam))`
- Is mogelijk met stored procedures.

```
CREATE PROCEDURE [dbo].[GetStudents]
    @FirstName varchar(50)
AS
BEGIN
    SET NOCOUNT ON;
    select * from Students where FirstName like @FirstName +'%'
END
```

⚔ SQL INJECTIE

- Is onmogelijk in onze ORM (EF Core) zolang we niet expliciet SQL schrijven
- Is mogelijk met `context.Database.ExecuteSqlCommand`:

  ❌ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = " + naam)`

  ❌ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = {0}", naam)`

  ✅ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = @x", new SqlParameter("@x", naam))`

- Is mogelijk met stored procedures.

```sql
CREATE PROCEDURE [dbo].[GetStudents]
    @FirstName varchar(50)
AS
BEGIN
    SET NOCOUNT ON;
    select * from Students where FirstName like @FirstName +'%'
END
```

⚔ SQL INJECTIE

- Is onmogelijk in onze ORM (EF Core) zolang we niet expliciet SQL schrijven
- Is mogelijk met `context.Database.ExecuteSqlCommand`:

  ✗ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = " + naam)`

  ✗ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = {0}", naam)`

  ☑ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = @x", new SqlParameter("@x", naam))`
- Is mogelijk met stored procedures.

```
CREATE PROCEDURE [dbo].[GetStudents]
    @FirstName varchar(50)
AS
BEGIN
    SET NOCOUNT ON;
    select * from Students where FirstName like @FirstName +'%'
END
```

⚔ SQL INJECTIE

- Is onmogelijk in onze ORM (EF Core) zolang we niet expliciet SQL schrijven
- Is mogelijk met `context.Database.ExecuteSqlCommand`:

✗ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = " + naam)`

✗ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = {0}", naam)`

☑ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = @x", new SqlParameter("@x", naam))`

- Is mogelijk met stored procedures.

```
CREATE PROCEDURE [dbo].[GetStudents]
    @FirstName varchar(50)
AS
BEGIN
    SET NOCOUNT ON;
    select * from Students where FirstName like @FirstName +'%'
END
```

⚔ SQL INJECTIE

- Is onmogelijk in onze ORM (EF Core) zolang we niet expliciet SQL schrijven
- Is mogelijk met `context.Database.ExecuteSqlCommand`:

✗ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = " + naam)`

✗ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = {0}", naam)`

☑ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = @x", new SqlParameter("@x", naam))`

- Is mogelijk met stored procedures.

```
CREATE PROCEDURE [dbo].[GetStudents]
    @FirstName varchar(50)
AS
BEGIN
    SET NOCOUNT ON;
    select * from Students where FirstName like @FirstName +'%'
END
```

⚔ SQL INJECTIE

- Is onmogelijk in onze ORM (EF Core) zolang we niet expliciet SQL schrijven
- Is mogelijk met `context.Database.ExecuteSqlCommand`:

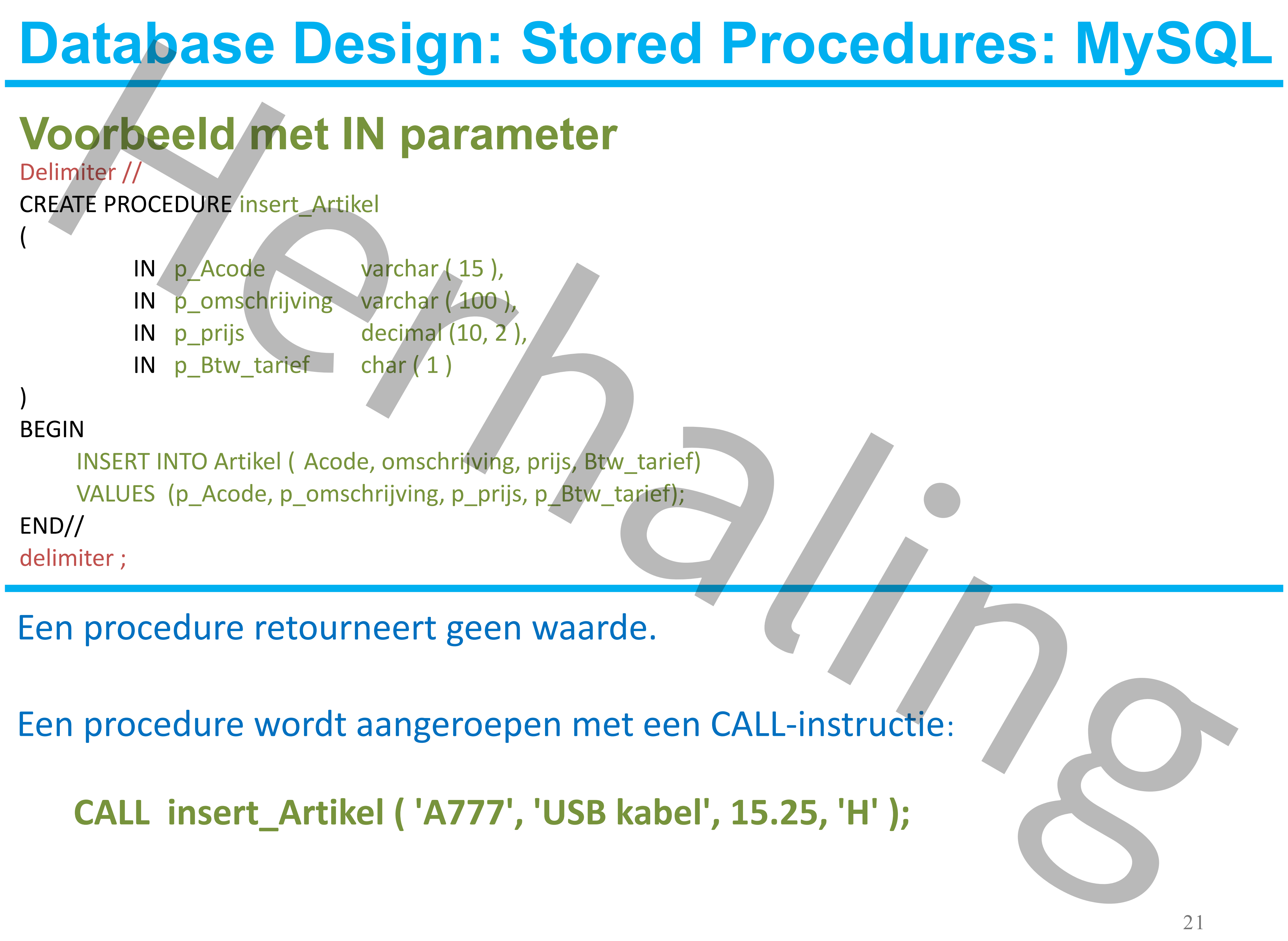
  ✗ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = " + naam)`

  ✗ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = {0}", naam)`

  ☑ `.ExecuteSqlCommand("SELECT * From Stud Where Naam = @x", new SqlParameter("@x", naam))`
- Is mogelijk met stored procedures.

```
CREATE PROCEDURE [dbo].[GetStudents]
    @FirstName varchar(50)
AS
BEGIN
    SET NOCOUNT ON;
    select * from Students where FirstName like @FirstName +'%'
END
```

STORED PROCEDURES



Database Design: Stored Procedures: MySQL

Voorbeeld met IN parameter

```
Delimiter //
CREATE PROCEDURE insert_Artikel
(
        IN  p_Acode          varchar ( 15 ),
        IN  p_omschrijving   varchar ( 100 ),
        IN  p_prijs          decimal (10, 2 ),
        IN  p_Btw_tarief     char ( 1 )
)
BEGIN
    INSERT INTO Artikel (  Acode, omschrijving, prijs, Btw_tarief)
    VALUES  (p_Acode, p_omschrijving, p_prijs, p_Btw_tarief);
END//
delimiter ;
```

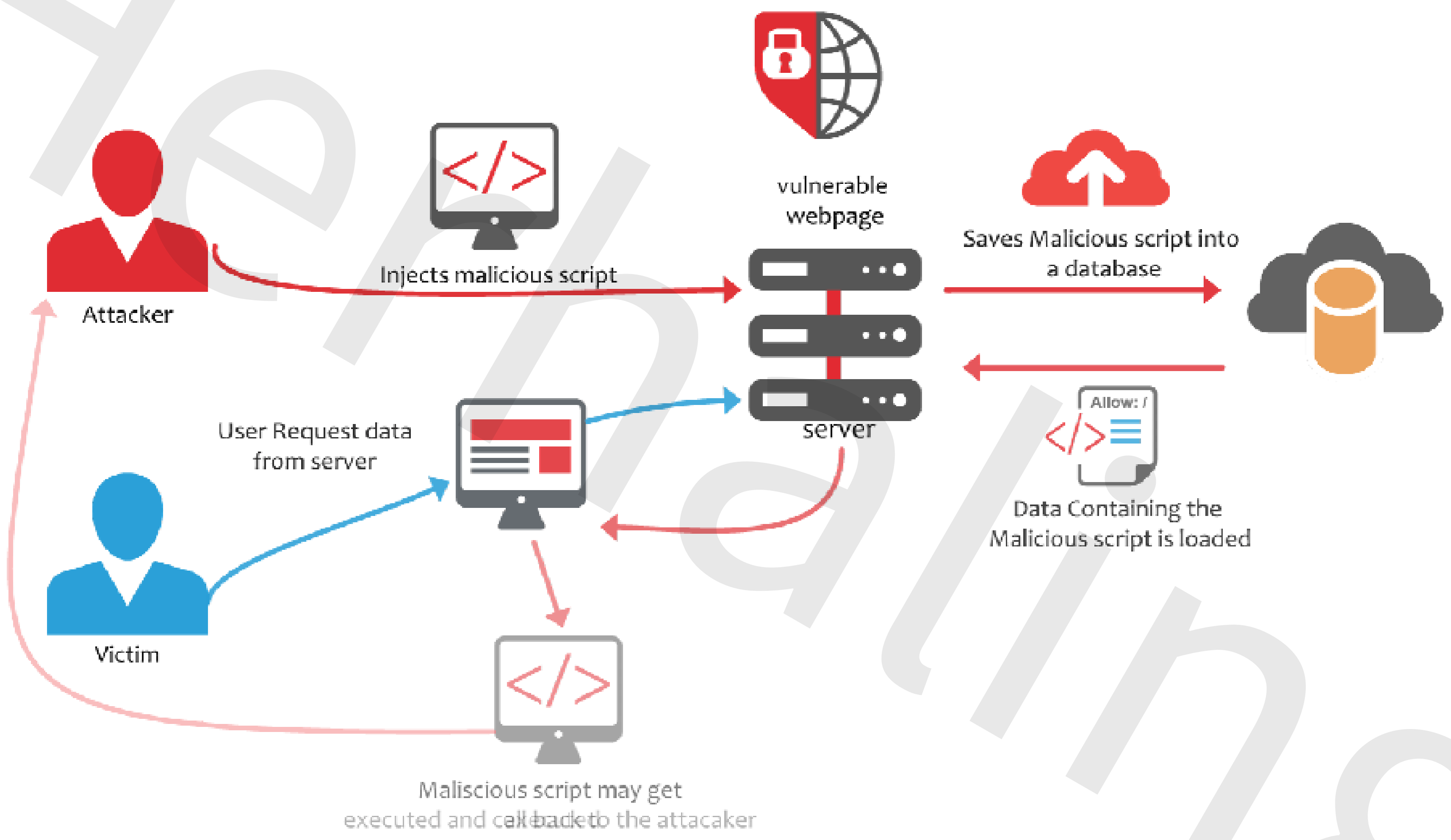
Een procedure retourneert geen waarde.

Een procedure wordt aangeroepen met een CALL-instructie:

```
CALL  insert_Artikel ( 'A777', 'USB kabel', 15.25, 'H' );
```

⚔ XSS

Bron: sheets SIE

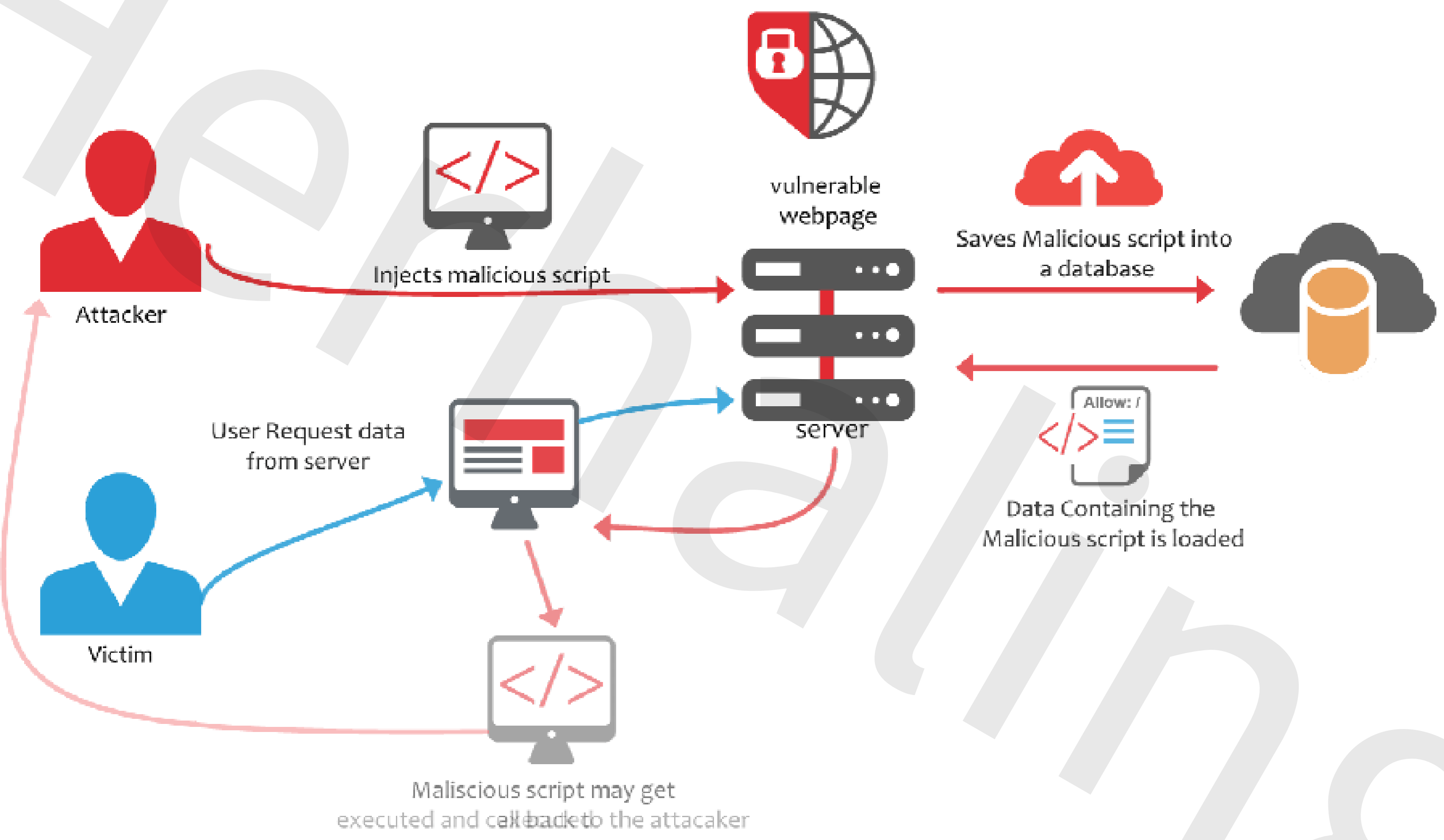

Attacker
Injects malicious script
vulnerable webpage
Saves Malicious script into a database
server
Allow: /
Data Containing the Malicious script is loaded
User Request data from server
Victim
Maliscious script may get executed and call back to the attacaker

NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

⚔ XSS

Bron: sheets SIE

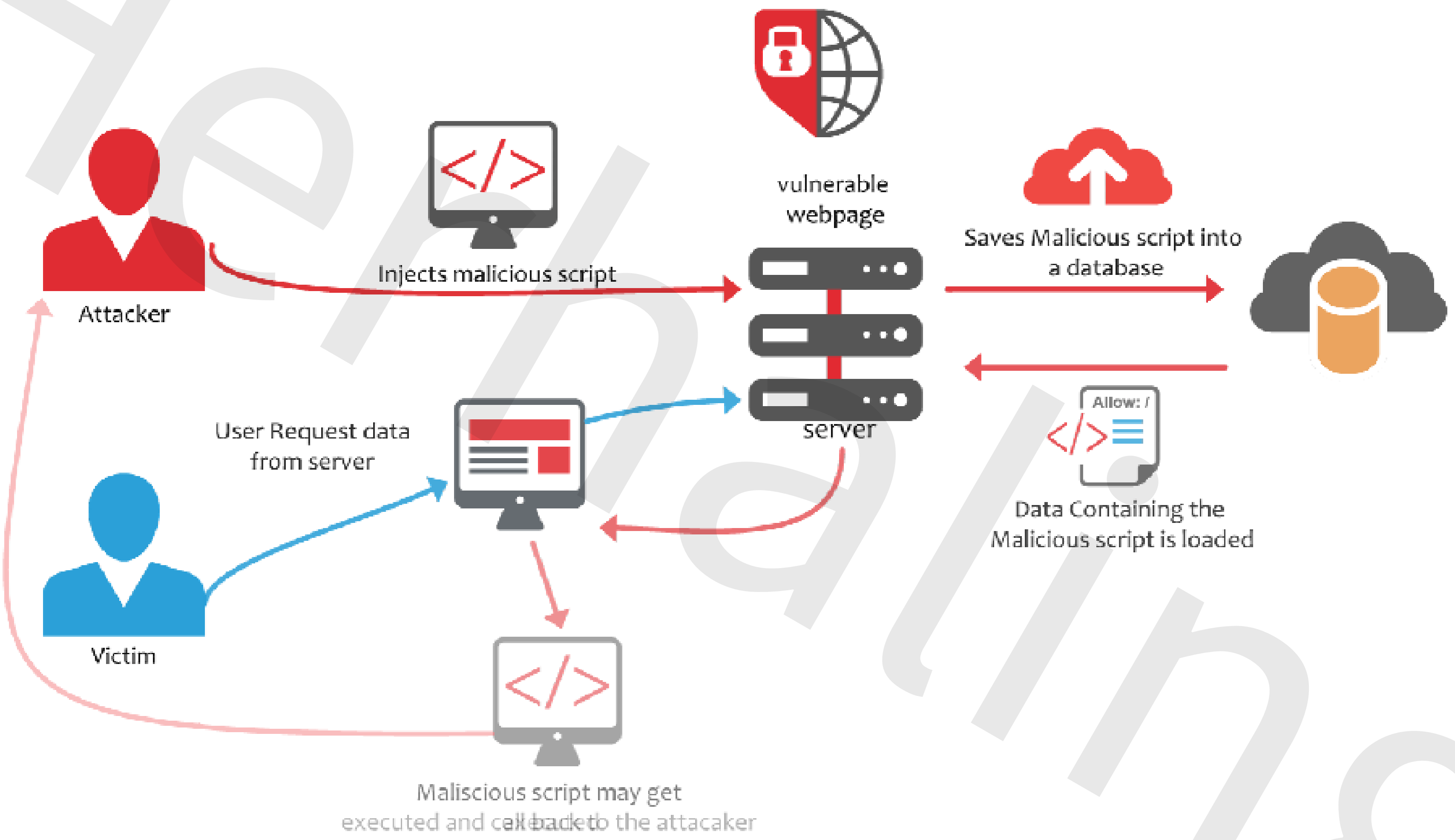

vulnerable webpage
Attacker
Injects malicious script
Saves Malicious script into a database
server
User Request data from server
Allow: /
Data Containing the Malicious script is loaded
Victim
Maliscious script may get executed and call back to the attacaker

NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

⚔ XSS

Bron: sheets SIE



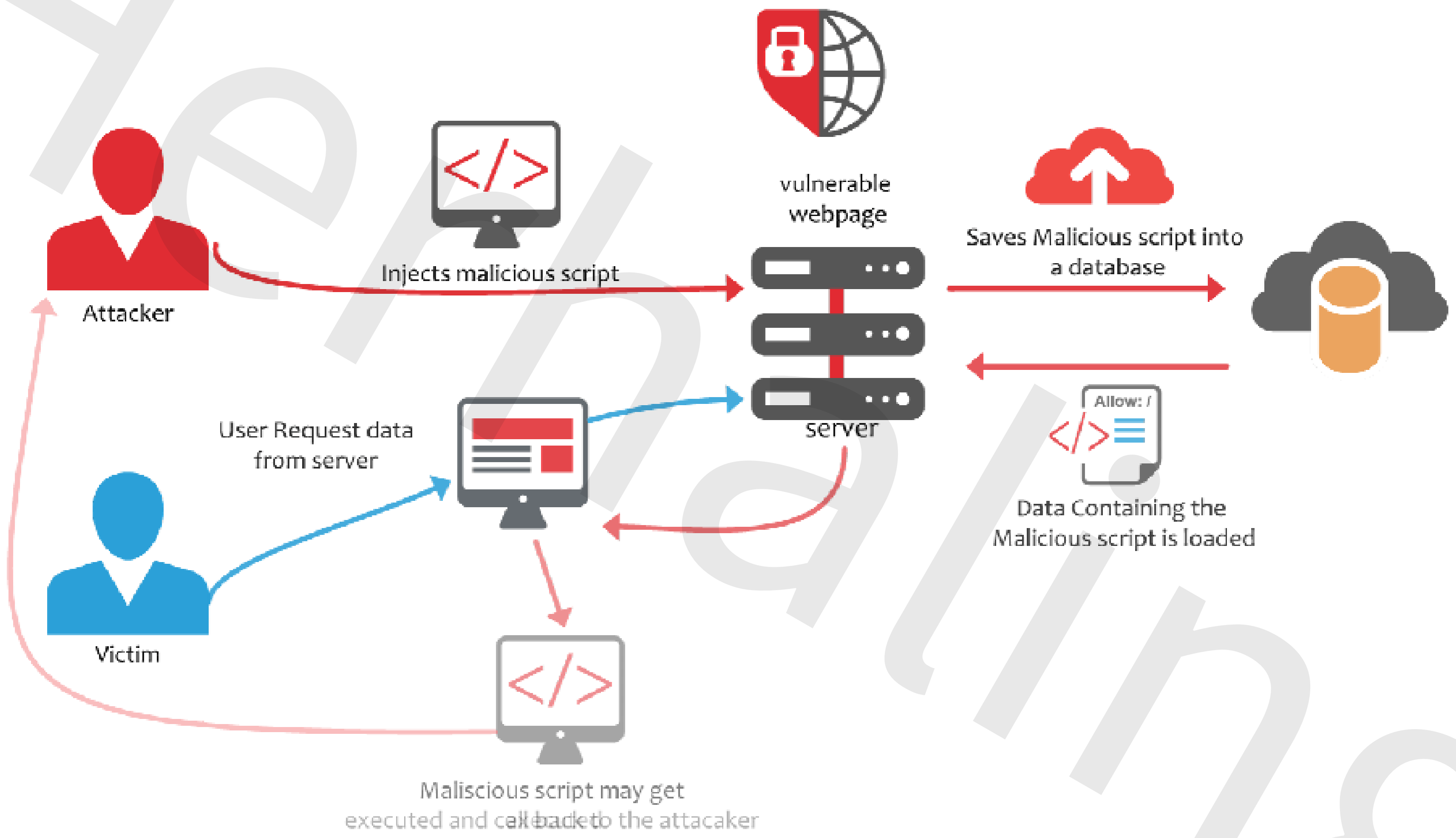
Attacker
Injects malicious script
vulnerable webpage
server
Saves Malicious script into a database
Allow: /
Data Containing the Malicious script is loaded
User Request data from server
Victim
Maliscious script may get executed and callback to the attacaker

NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

⚔ XSS

Bron: sheets SIE

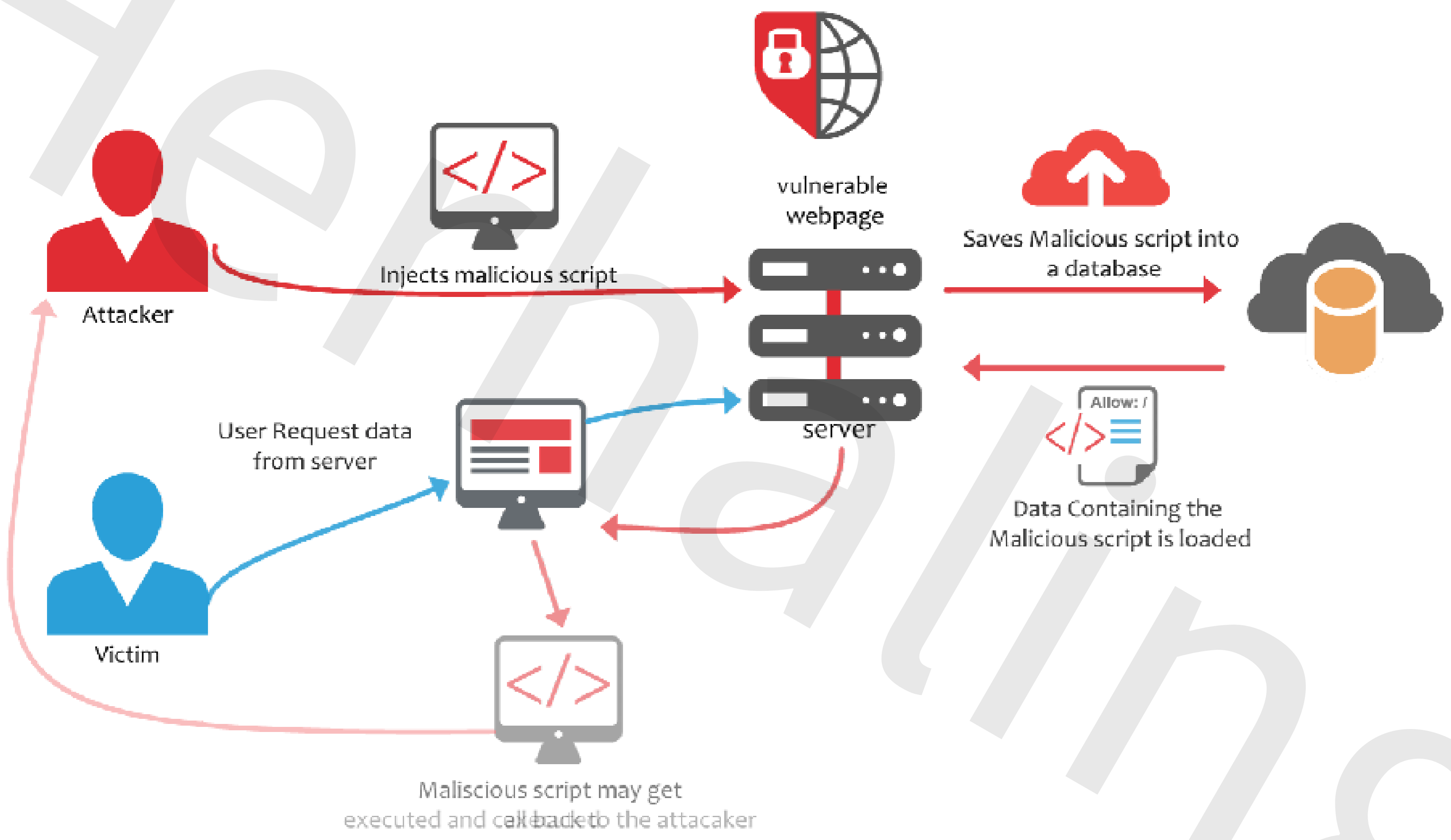

Attacker
Injects malicious script
vulnerable webpage
Saves Malicious script into a database
server
Allow: /
Data Containing the Malicious script is loaded
User Request data from server
Victim
Maliscious script may get executed and call back to the attacaker

NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

⚔ XSS

Bron: sheets SIE

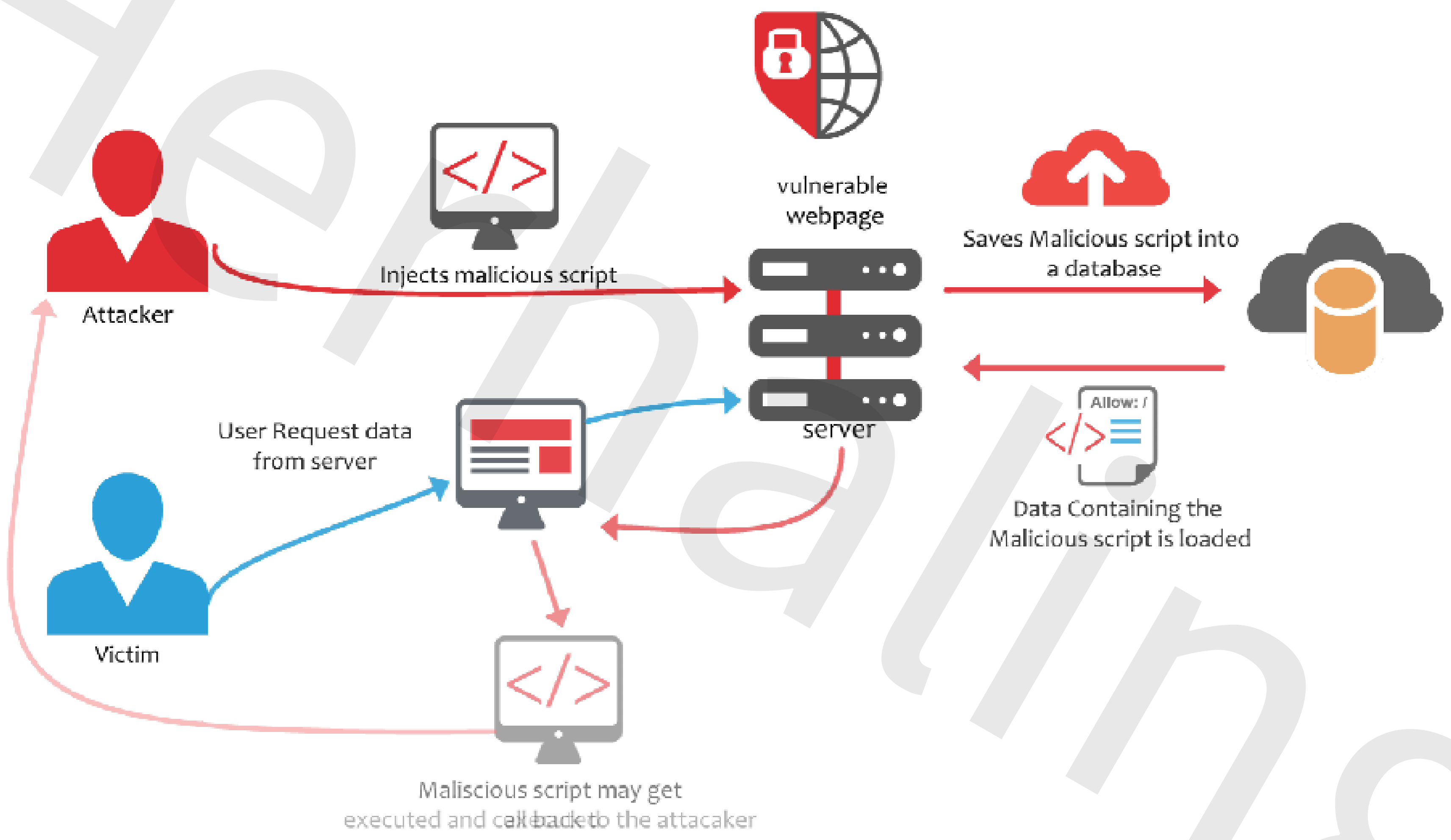

vulnerable webpage
Attacker
Injects malicious script
Saves Malicious script into a database
server
User Request data from server
Allow: /
Data Containing the Malicious script is loaded
Victim
Maliscious script may get executed and call back to the attacaker

NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

⚔ XSS

Bron: sheets SIE

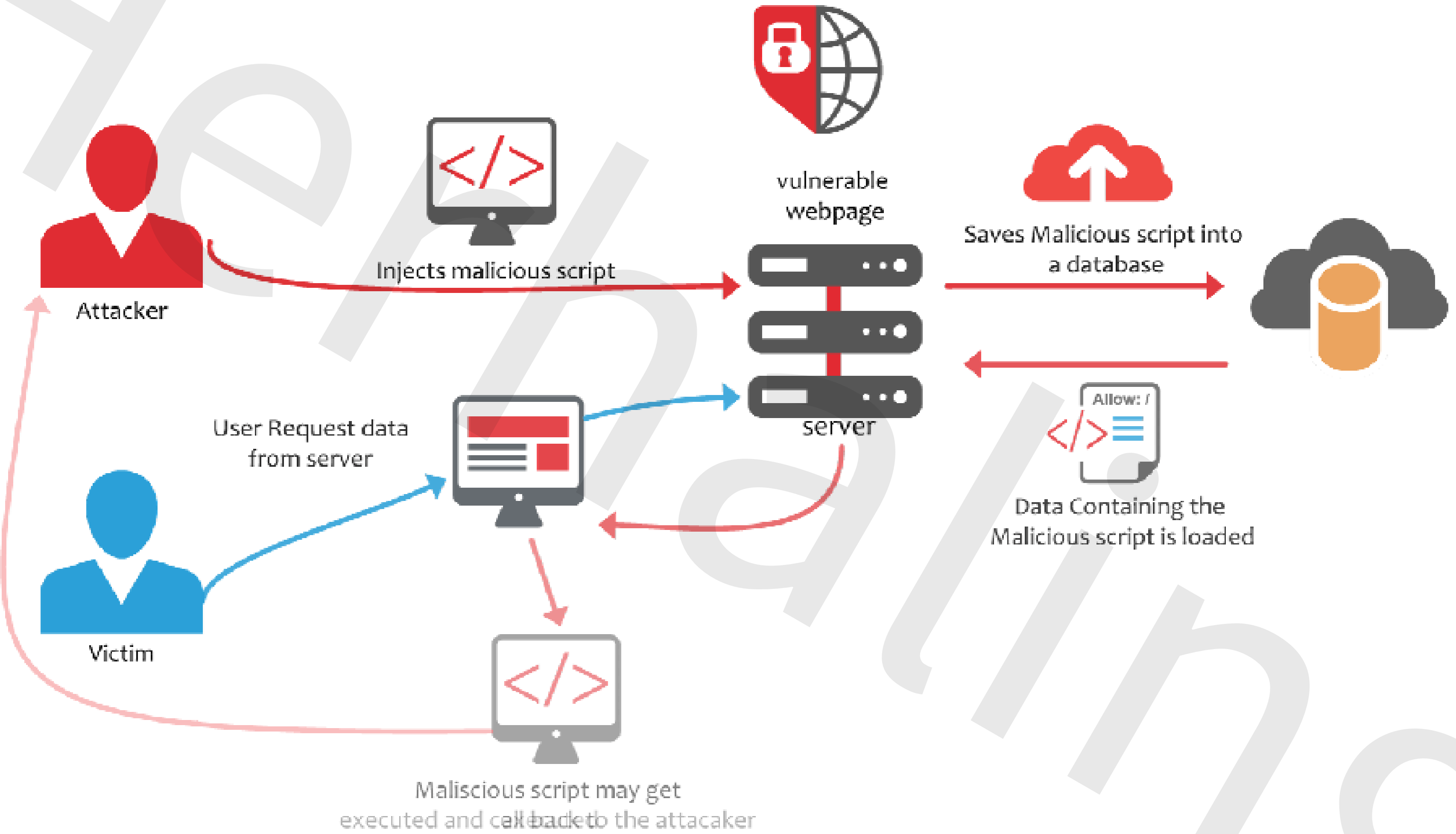

vulnerable webpage
Attacker
Injects malicious script
Saves Malicious script into a database
server
User Request data from server
Allow: /
Data Containing the Malicious script is loaded
Victim
Maliscious script may get executed and call back to the attacaker

NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

⚔ XSS

Bron: sheets SIE

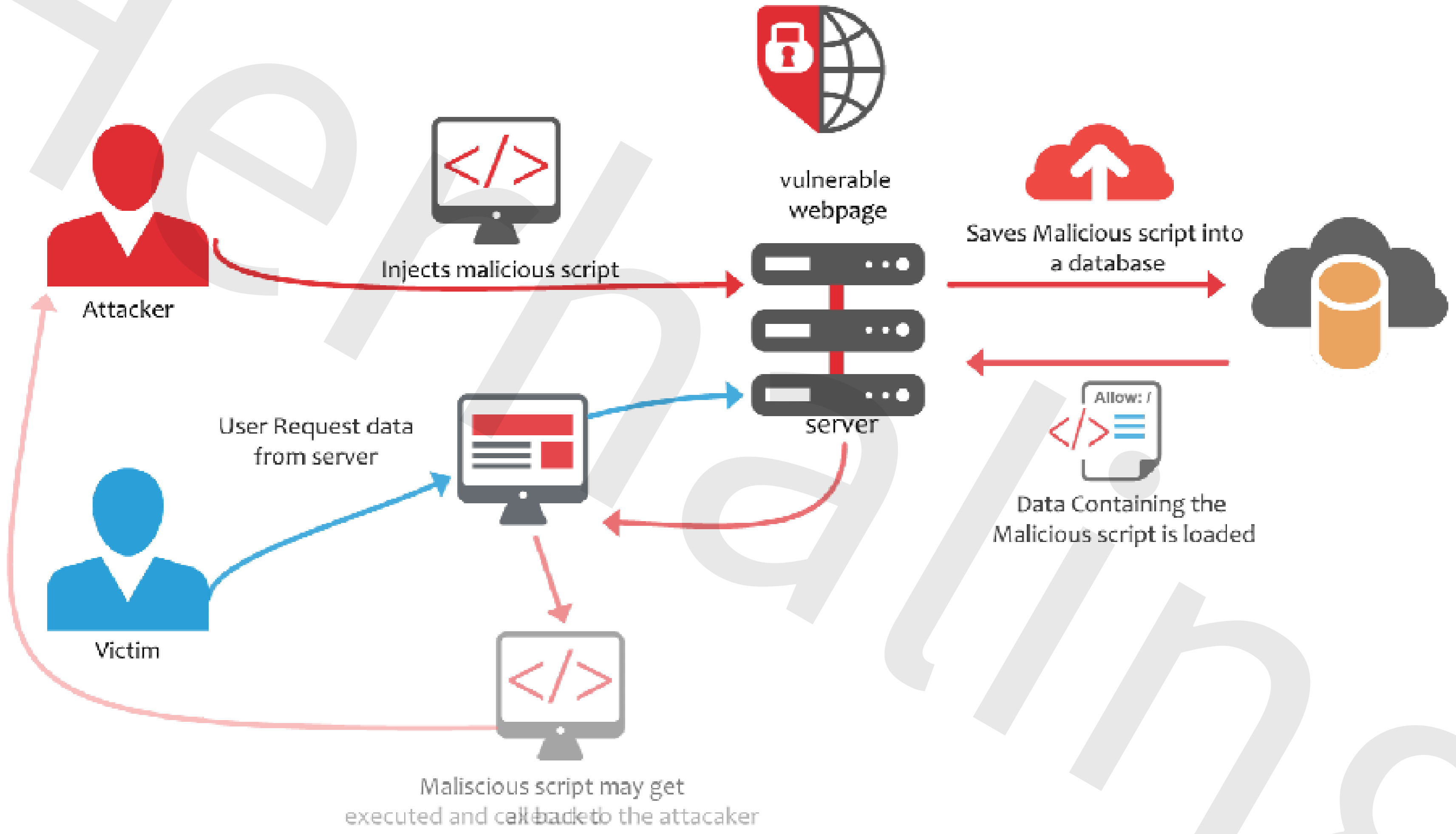


NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

⚔ XSS

Bron: sheets SIE

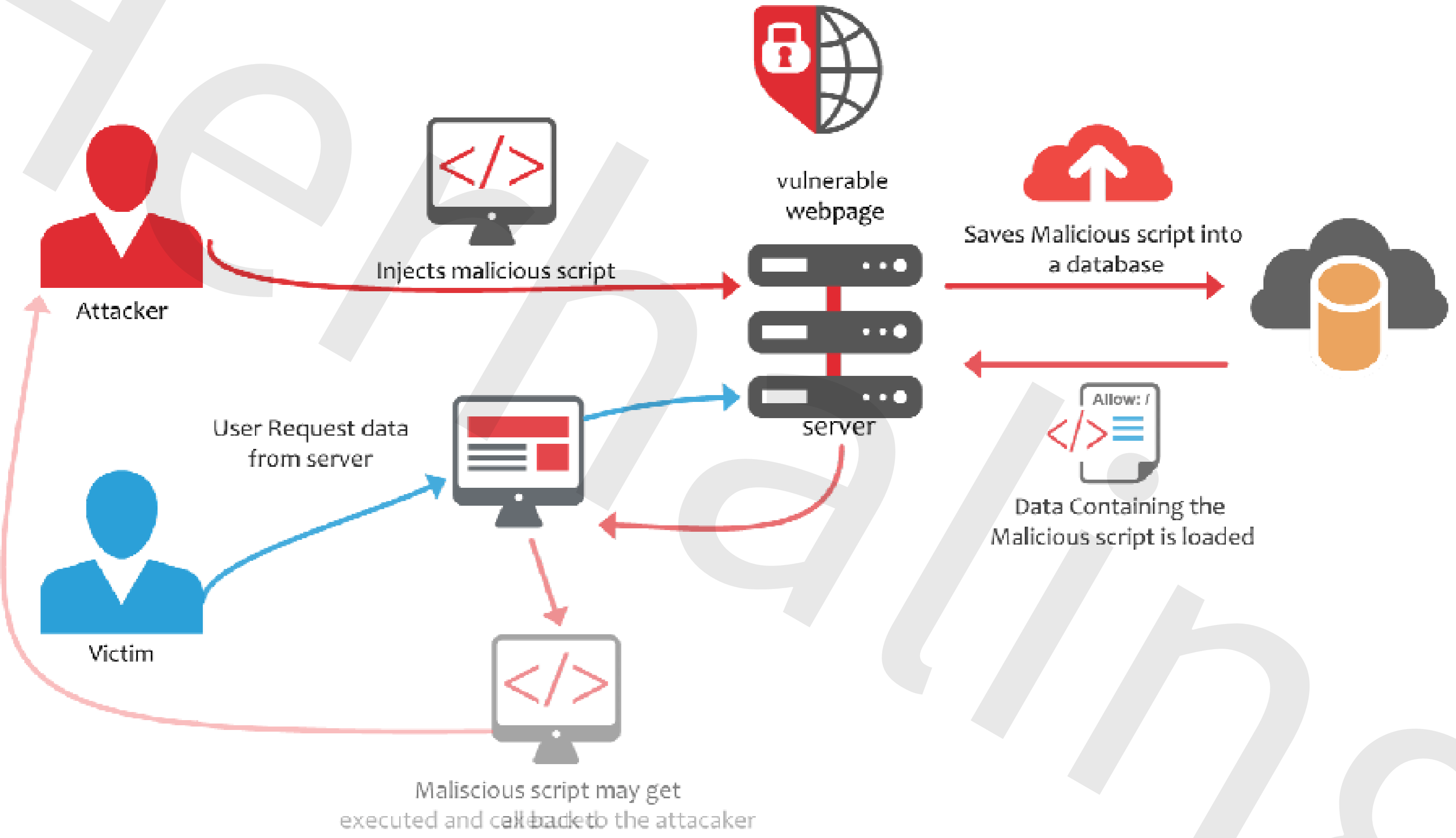

vulnerable webpage
Attacker
Injects malicious script
Saves Malicious script into a database
server
User Request data from server
Allow: /
Data Containing the Malicious script is loaded
Victim
Maliscious script may get executed and call back to the attacaker

NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

⚔ XSS

Bron: sheets SIE



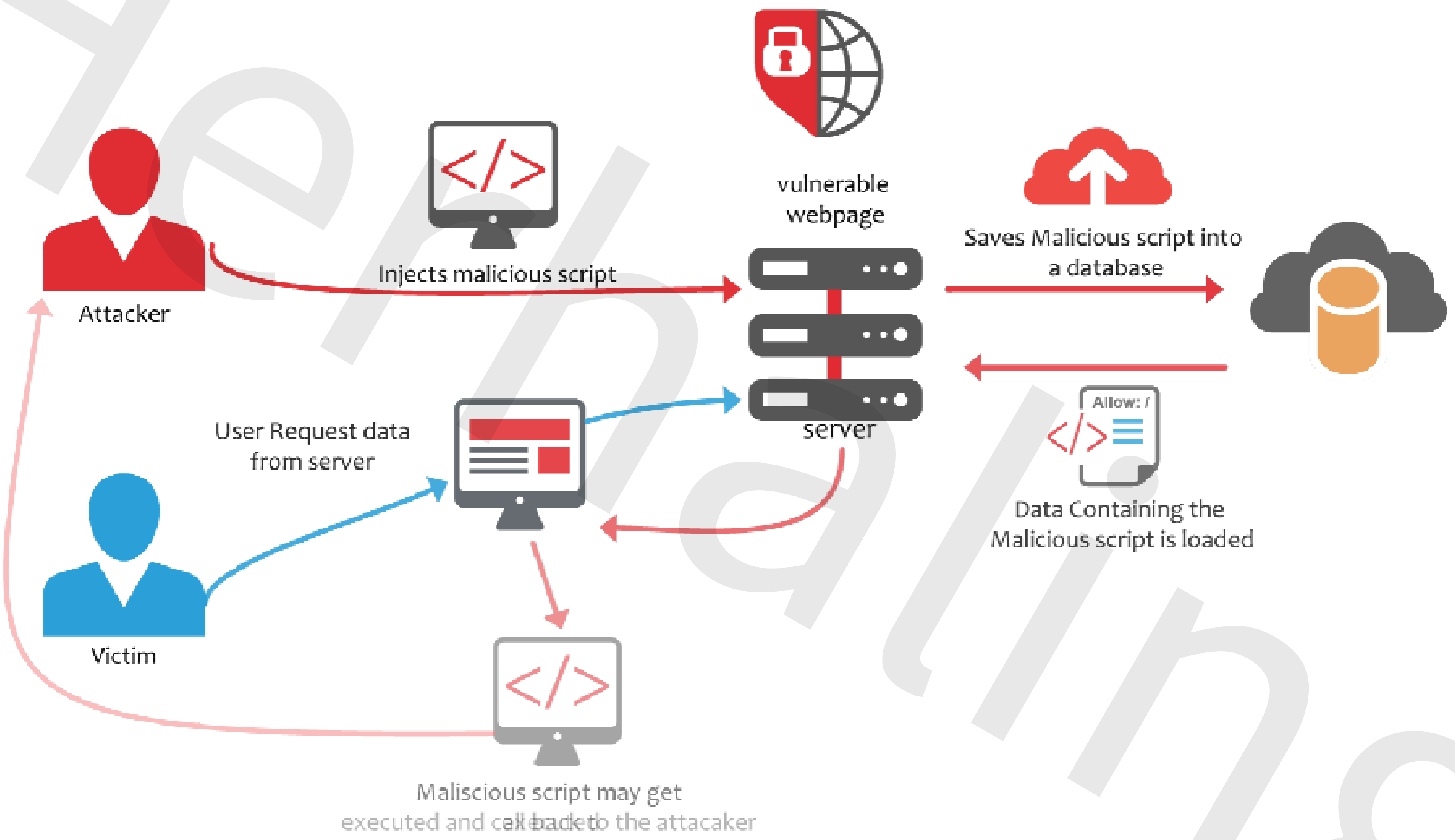
Attacker
Injects malicious script
vulnerable webpage
Saves Malicious script into a database
server
Allow: /
Data Containing the Malicious script is loaded
User Request data from server
Victim
Maliscious script may get executed and call back to the attacaker

NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

⚔ XSS

Bron: sheets SIE



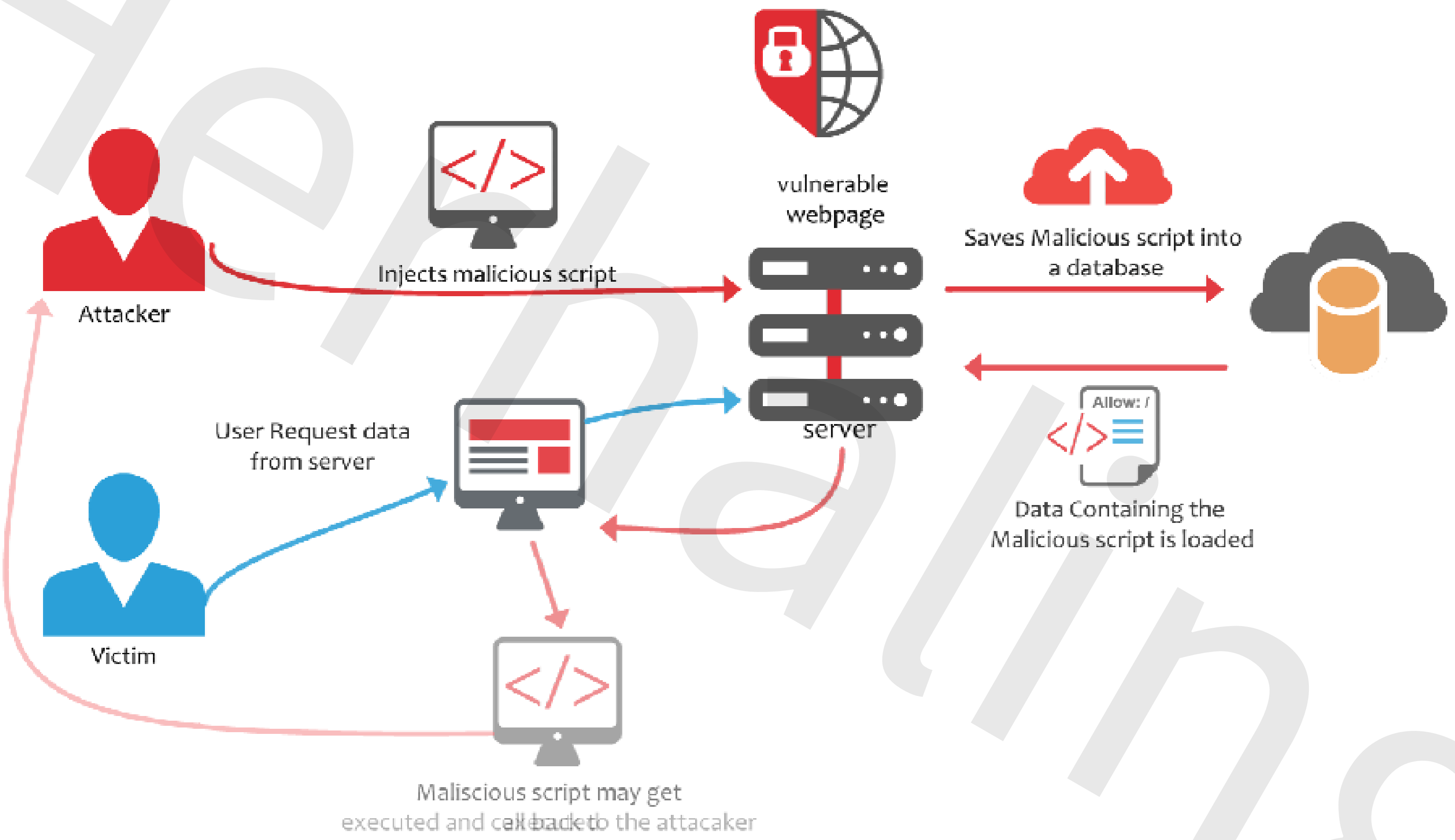
vulnerable webpage
Attacker
Injects malicious script
Saves Malicious script into a database
server
User Request data from server
Allow: /
Data Containing the Malicious script is loaded
Victim
Maliscious script may get executed and call back to the attacaker

NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

⚔ XSS

Bron: sheets SIE


Attacker
Injects malicious script
vulnerable webpage
server
Saves Malicious script into a database
Allow: /
Data Containing the Malicious script is loaded
User Request data from server
Victim
Maliscious script may get executed and call back to the attacaker

NB: de code hoeft niet opgeslagen te zijn in de database.

Nu een demo met ASP.NET Core: localhost:8080/Students/Zoek?Bob

🛡 XSS IS BIJNA NIET MOGELIJK IN ASP.NET CORE

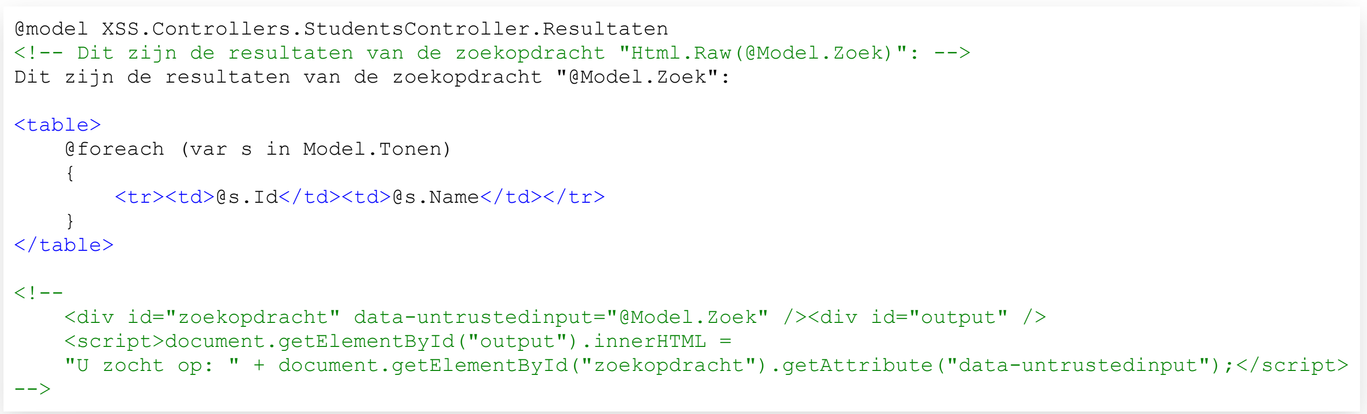
Zie de demo voor uitleg.

```
@model XSS.Controllers.StudentsController.Resultaten
<!-- Dit zijn de resultaten van de zoekopdracht "Html.Raw(@Model.Zoek)": -->
Dit zijn de resultaten van de zoekopdracht "@Model.Zoek":

<table>
    @foreach (var s in Model.Tonen)
    {
        <tr><td>@s.Id</td><td>@s.Name</td></tr>
    }
</table>

<!--
    <div id="zoekopdracht" data-untrustedinput="@Model.Zoek" /><div id="output" />
    <script>document.getElementById("output").innerHTML =
    "U zocht op: " + document.getElementById("zoekopdracht").getAttribute("data-untrustedinput");</script>
-->
```

- Veilig, tenzij je de standaard encoders niet gebruikt (die zetten < om in `&lt;`), door bijv. `HtmlHelper.Raw`

🛡 XSS IS BIJNA NIET MOGELIJK IN ASP.NET CORE

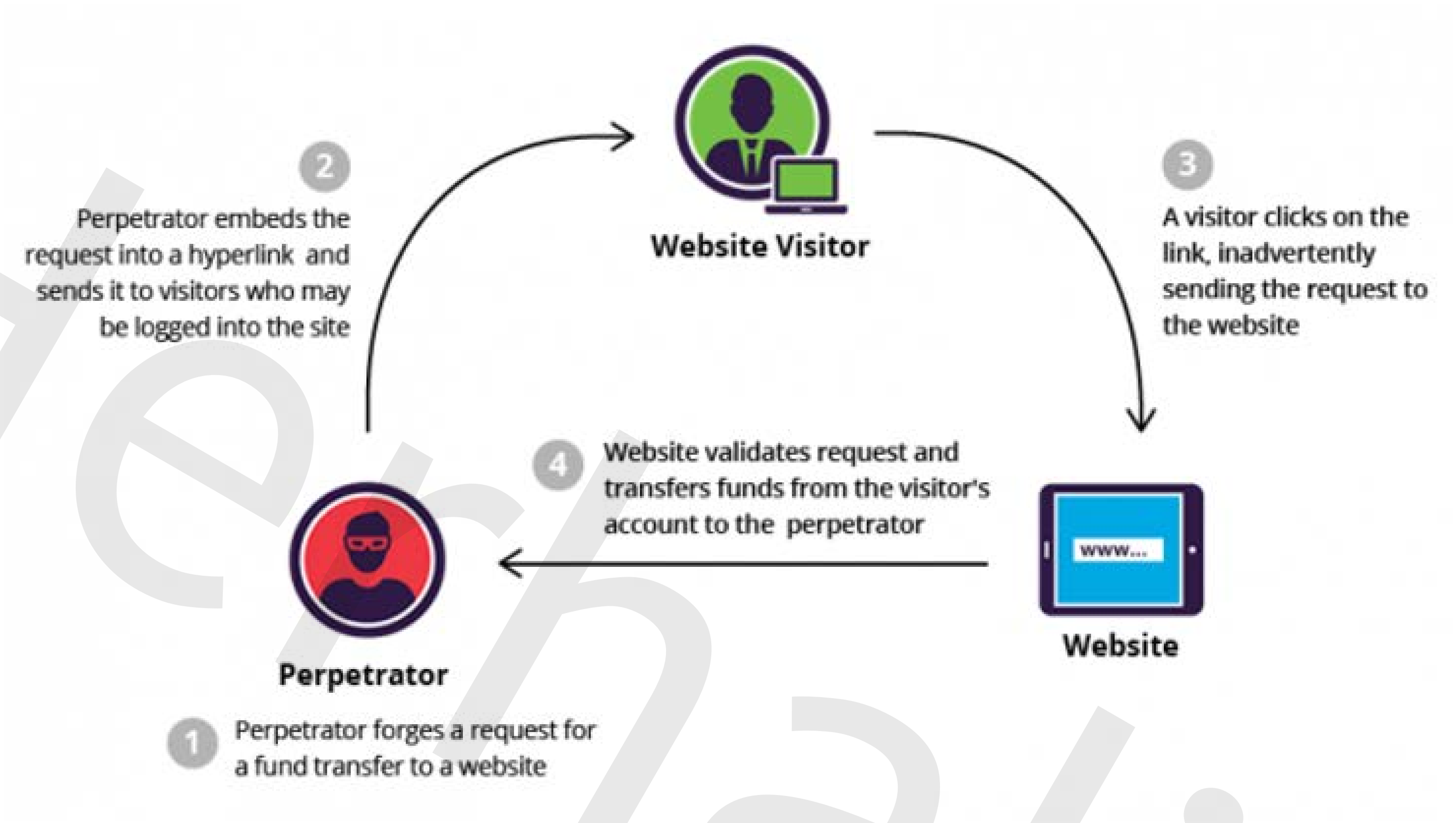
Zie de demo voor uitleg.



```
@model XSS.Controllers.StudentsController.Resultaten
<!-- Dit zijn de resultaten van de zoekopdracht "Html.Raw(@Model.Zoek)": -->
Dit zijn de resultaten van de zoekopdracht "@Model.Zoek":

<table>
    @foreach (var s in Model.Tonen)
    {
        <tr><td>@s.Id</td><td>@s.Name</td></tr>
    }
</table>

<!--
    <div id="zoekopdracht" data-untrustedinput="@Model.Zoek" /><div id="output" />
    <script>document.getElementById("output").innerHTML =
    "U zocht op: " + document.getElementById("zoekopdracht").getAttribute("data-untrustedinput");</script>
-->
```

- Veilig, tenzij je de standaard encoders niet gebruikt (die zetten `<` om in `&lt;`), door bijv. `HtmlHelper.Raw`

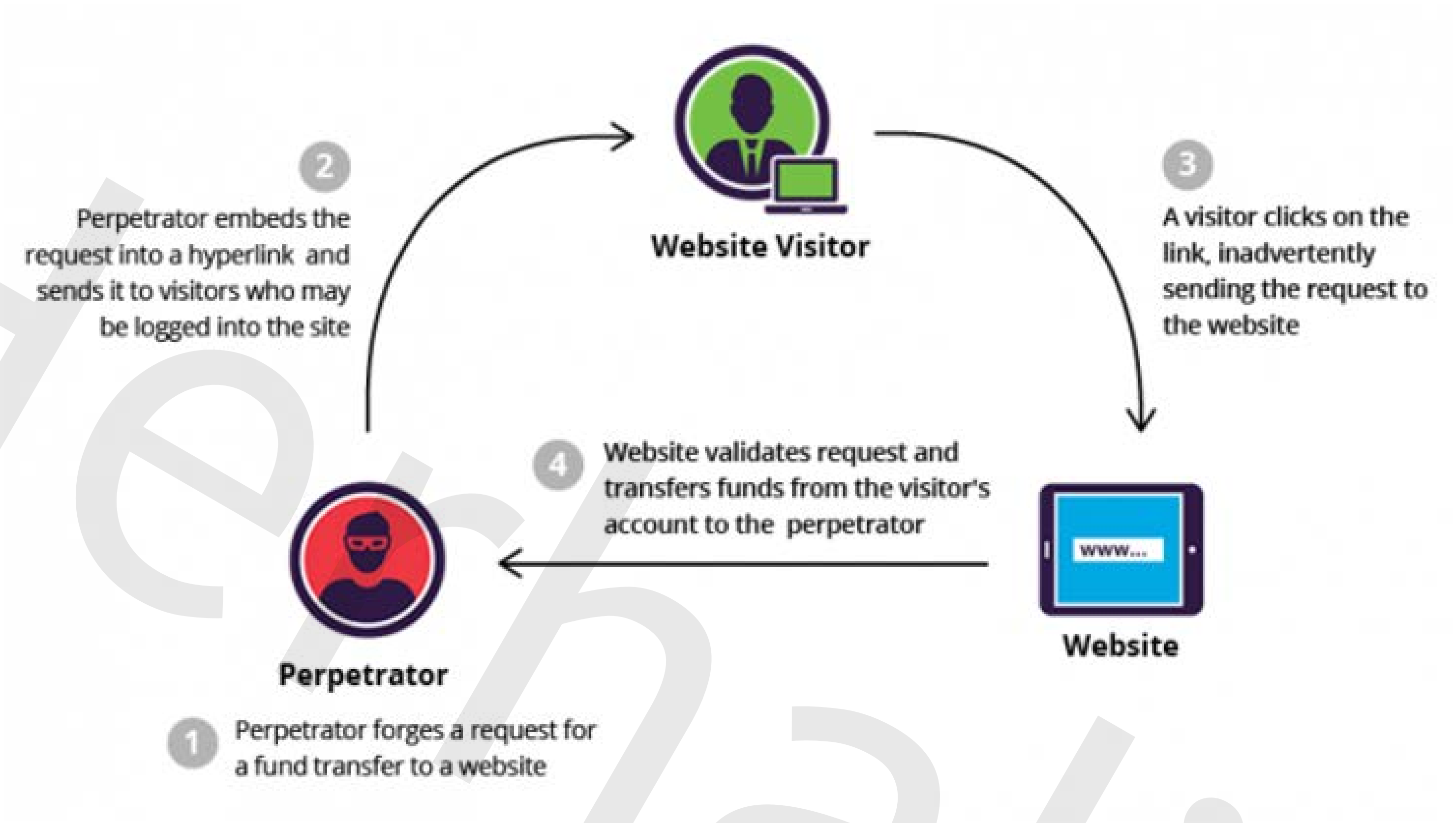
XSRF


Website Visitor
2
Perpetrator embeds the request into a hyperlink and sends it to visitors who may be logged into the site
3
A visitor clicks on the link, inadvertently sending the request to the website
4
Website validates request and transfers funds from the visitor's account to the perpetrator
www...
Website
Perpetrator
1
Perpetrator forges a request for a fund transfer to a website

Kan ook met POST requests:

```
<h1>Congratulations! You're a Winner!</h1>
<form action="http://rabobank.nl/api/account" method="post">
    <input type="hidden" name="Transaction" value="withdraw">
    <input type="hidden" name="Amount" value="1000000">
    <input type="submit" value="Click to collect your prize!">
</form>
```

⚔ XSRF



1 Perpetrator forges a request for a fund transfer to a website

2 Perpetrator embeds the request into a hyperlink and sends it to visitors who may be logged into the site

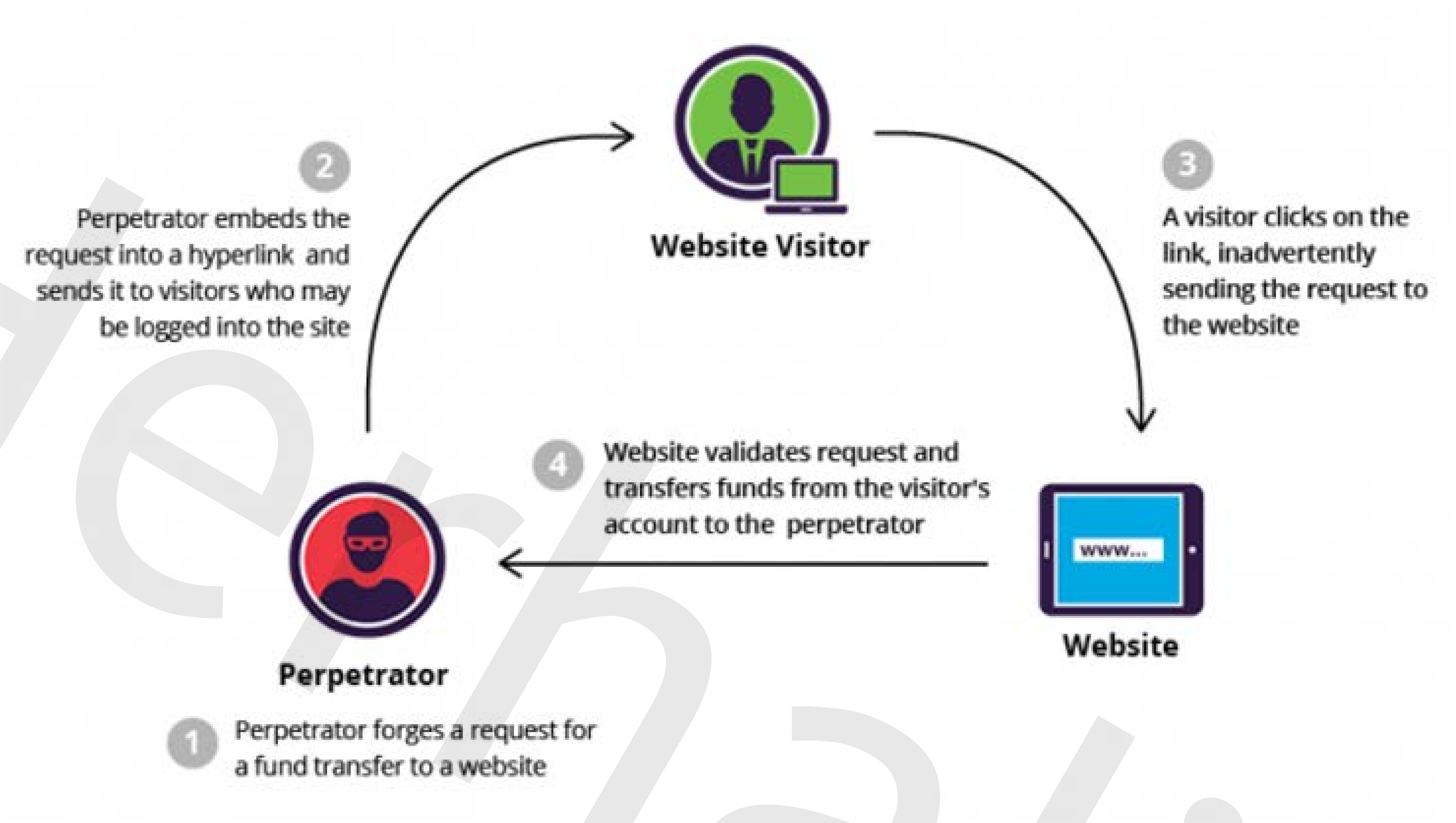
3 A visitor clicks on the link, inadvertently sending the request to the website

4 Website validates request and transfers funds from the visitor's account to the perpetrator

Kan ook met POST requests:

```
<h1>Congratulations! You're a Winner!</h1>
<form action="http://rabobank.nl/api/account" method="post">
    <input type="hidden" name="Transaction" value="withdraw">
    <input type="hidden" name="Amount" value="1000000">
    <input type="submit" value="Click to collect your prize!">
</form>
```

⚔ XSRF


Website Visitor
2
Perpetrator embeds the request into a hyperlink and sends it to visitors who may be logged into the site
3
A visitor clicks on the link, inadvertently sending the request to the website
4
Website validates request and transfers funds from the visitor's account to the perpetrator
www...
Website
Perpetrator
1
Perpetrator forges a request for a fund transfer to a website

Kan ook met POST requests:

```
<h1>Congratulations! You're a Winner!</h1>
<form action="http://rabobank.nl/api/account" method="post">
    <input type="hidden" name="Transaction" value="withdraw">
    <input type="hidden" name="Amount" value="1000000">
    <input type="submit" value="Click to collect your prize!">
</form>
```

🛡 ANTIFORGERY TOKEN

- Bij het formulier wordt een server-side gegenereerde token meegestuurd
- Deze **antiforgery token** wordt in het formulier teruggestuurd

```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw">
```

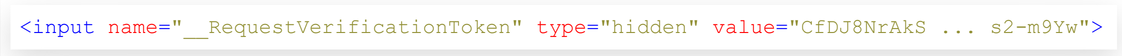
- Server-side wordt de token gecheckt met `[ValidateAntiForgeryToken]`:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Betaal(TransactieModel transactie)
{
    // ...
}
```

- Wordt automatisch toegevoegd (kijk maar eens in de HTML)
- Wordt niet gebruikt bij API calls

🛡 ANTIFORGERY TOKEN

- Bij het formulier wordt een server-side gegenereerde token meegestuurd
- Deze **antiforgery token** wordt in het formulier teruggestuurd



```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw">
```

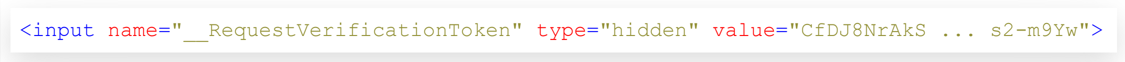
- Server-side wordt de token gecheckt met `[ValidateAntiForgeryToken]`:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Betaal(TransactieModel transactie)
{
    // ...
}
```

- Wordt automatisch toegevoegd (kijk maar eens in de HTML)
- Wordt niet gebruikt bij API calls

🛡 ANTIFORGERY TOKEN

- Bij het formulier wordt een server-side gegenereerde token meegestuurd
- Deze **antiforgery token** wordt in het formulier teruggestuurd



```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw">
```

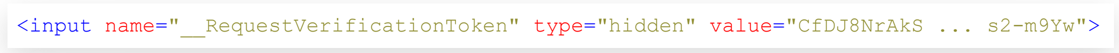
- Server-side wordt de token gecheckt met `[ValidateAntiForgeryToken]`:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Betaal(TransactieModel transactie)
{
    // ...
}
```

- Wordt automatisch toegevoegd (kijk maar eens in de HTML)
- Wordt niet gebruikt bij API calls

🛡 ANTIFORGERY TOKEN

- Bij het formulier wordt een server-side gegenereerde token meegestuurd
- Deze **antiforgery token** wordt in het formulier teruggestuurd



```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw">
```

- Server-side wordt de token gecheckt met `[ValidateAntiForgeryToken]`:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Betaal(TransactieModel transactie)
{
    // ...
}
```

- Wordt automatisch toegevoegd (kijk maar eens in de HTML)
- Wordt niet gebruikt bij API calls

🛡 ANTIFORGERY TOKEN

- Bij het formulier wordt een server-side gegenereerde token meegestuurd
- Deze **antiforgery token** wordt in het formulier teruggestuurd

```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw">
```

- Server-side wordt de token gecheckt met `[ValidateAntiForgeryToken]`:

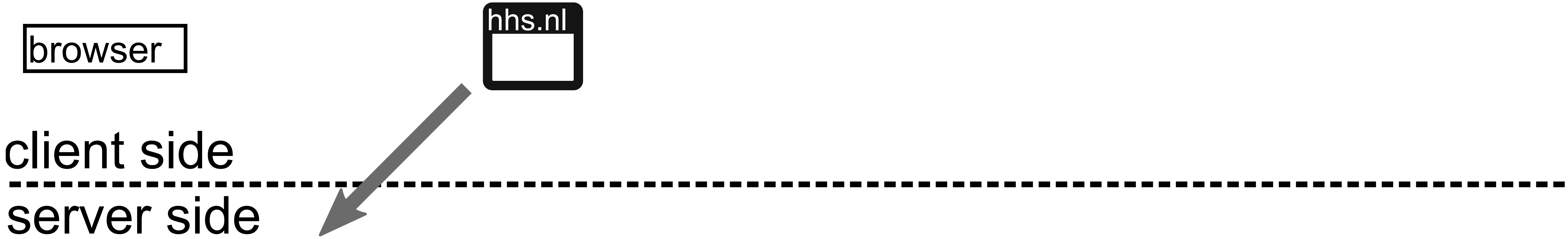
```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Betaal(TransactieModel transactie)
{
    // ...
}
```

- Wordt automatisch toegevoegd (kijk maar eens in de HTML)
- Wordt niet gebruikt bij API calls

🛡 SAME ORIGIN POLICY (VS CORS)

browser
hhs.nl

🛡 SAME ORIGIN POLICY (VS CORS)



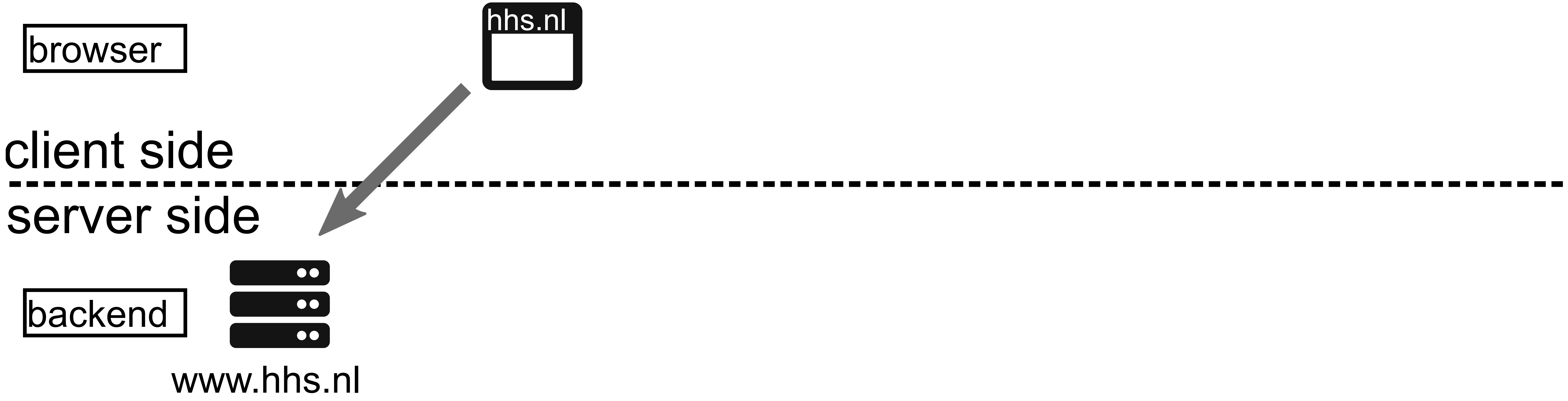
browser

hhs.nl

client side

server side

🛡 SAME ORIGIN POLICY (VS CORS)



hhs.nl

browser

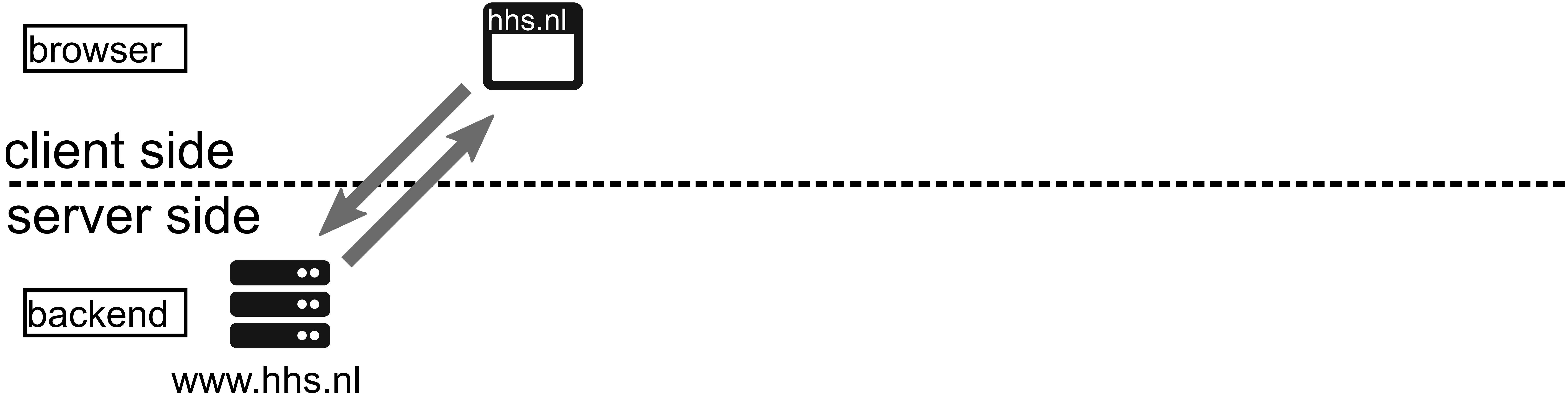
client side

server side

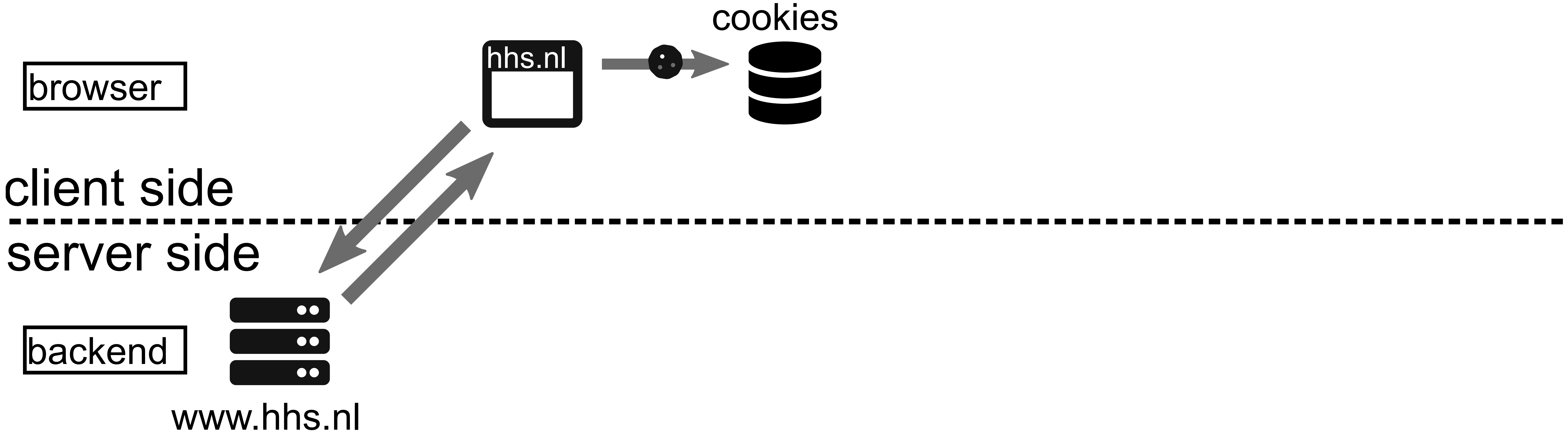
backend

www.hhs.nl

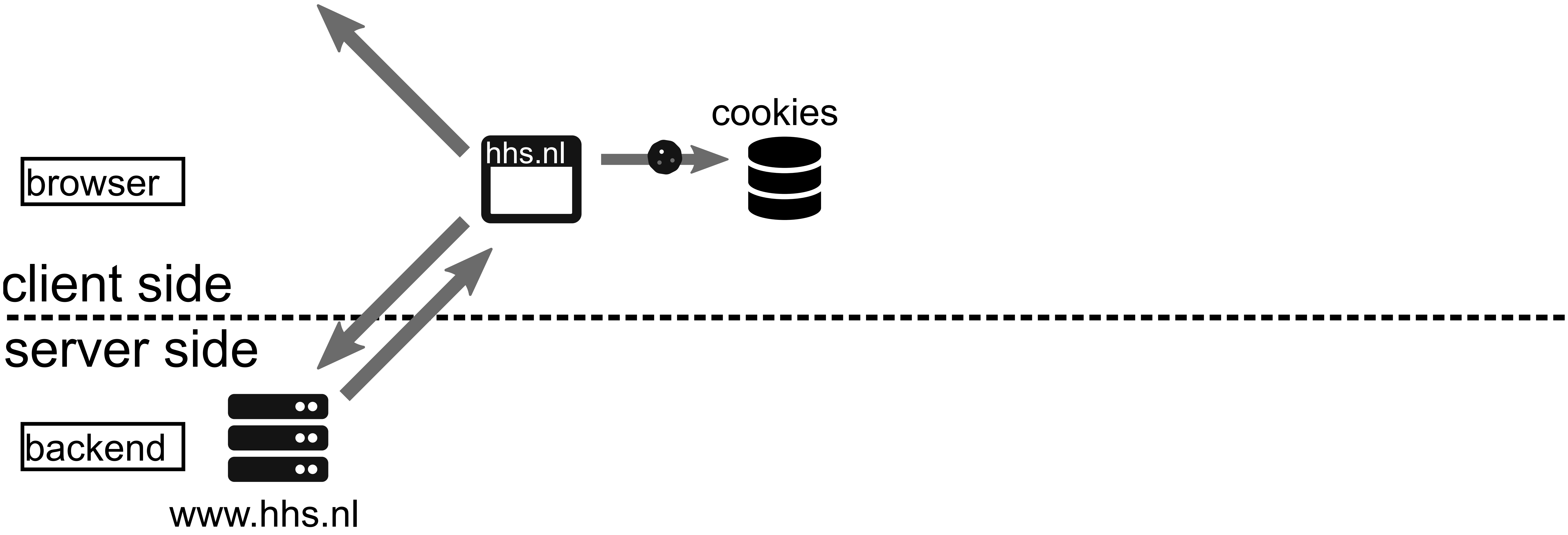
🛡 SAME ORIGIN POLICY (VS CORS)



🛡️ SAME ORIGIN POLICY (VS CORS)

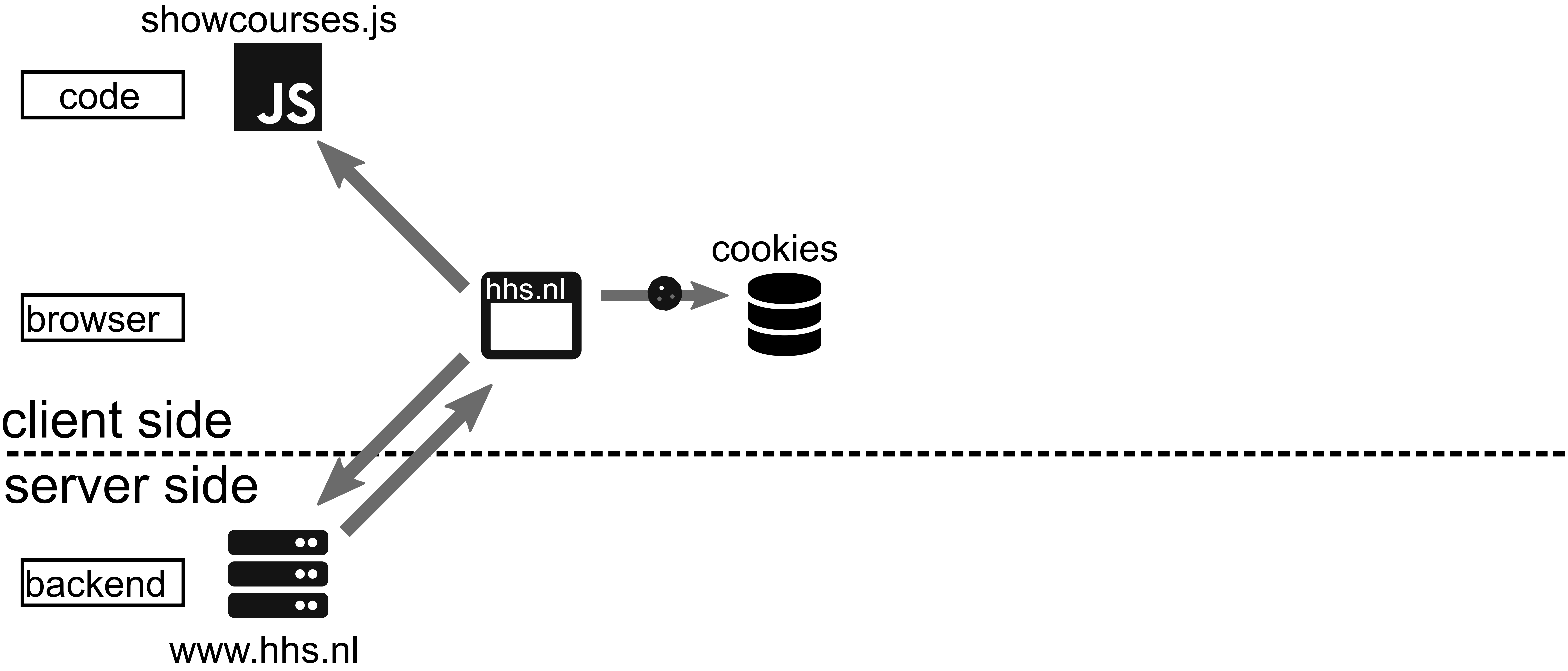


🛡 SAME ORIGIN POLICY (VS CORS)



≡

🛡 SAME ORIGIN POLICY (VS CORS)



showcourses.js

code

browser

hhs.nl

cookies

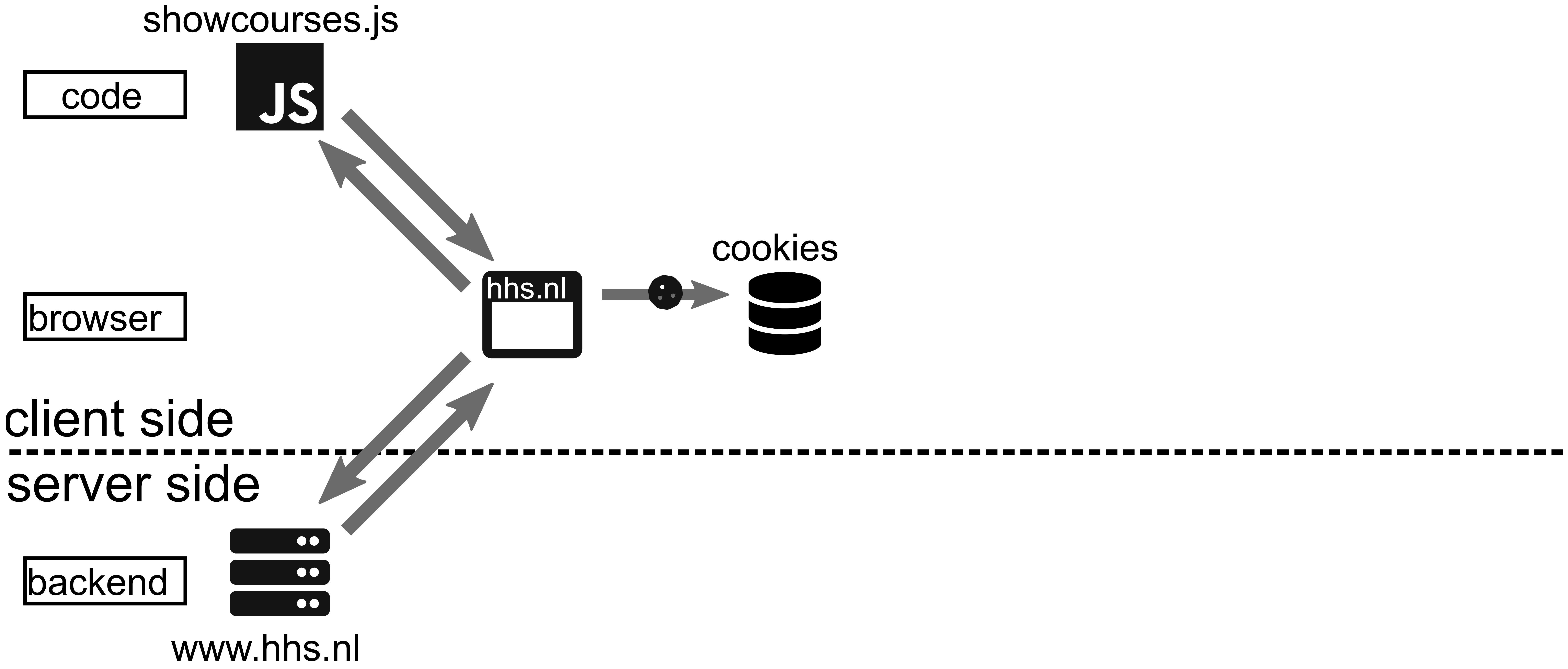
client side

server side

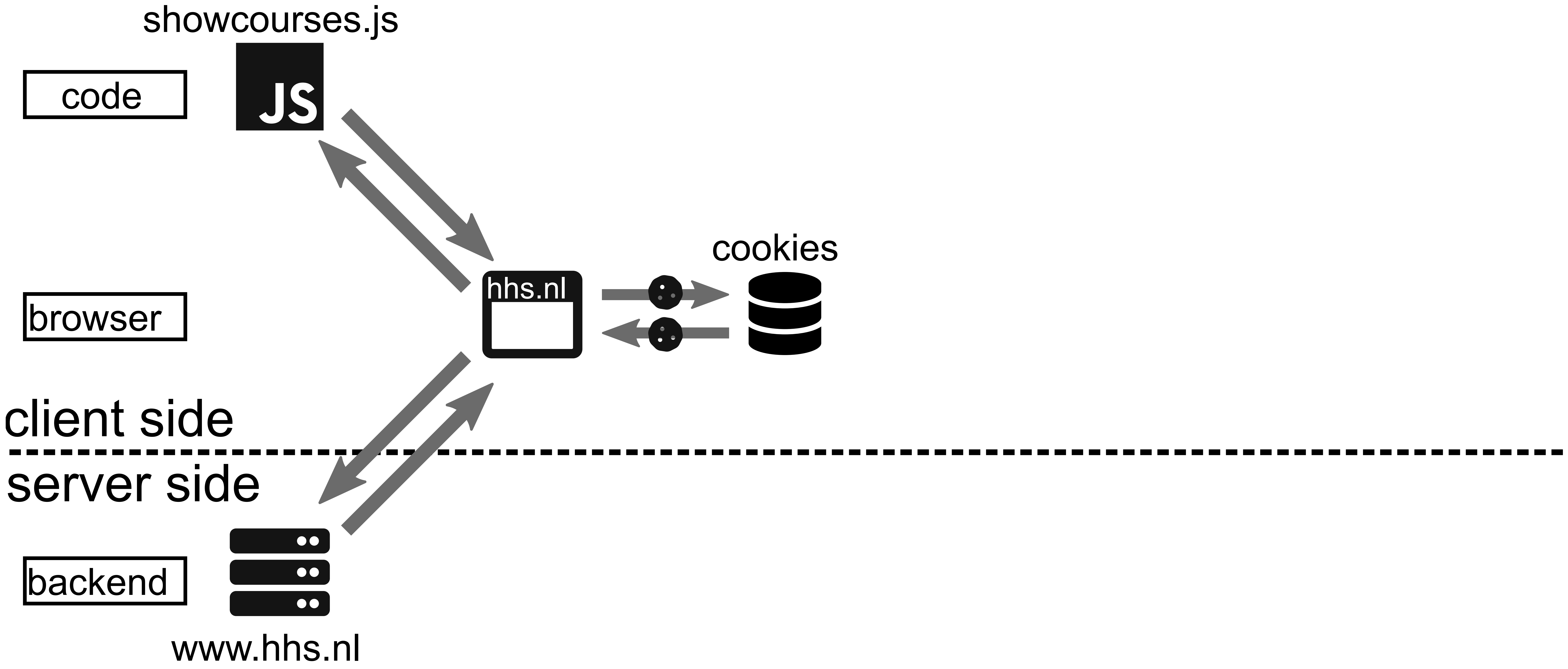
backend

www.hhs.nl

🛡 SAME ORIGIN POLICY (VS CORS)

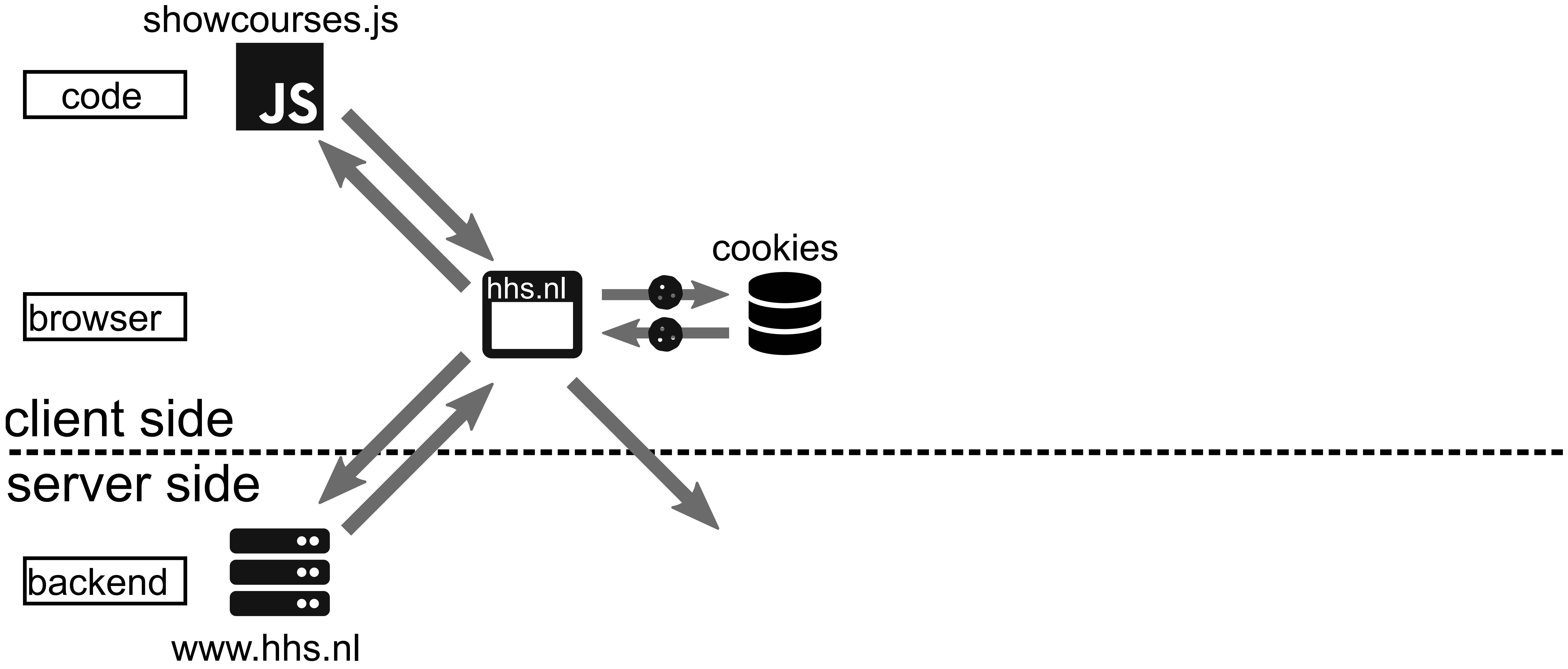


🛡 SAME ORIGIN POLICY (VS CORS)



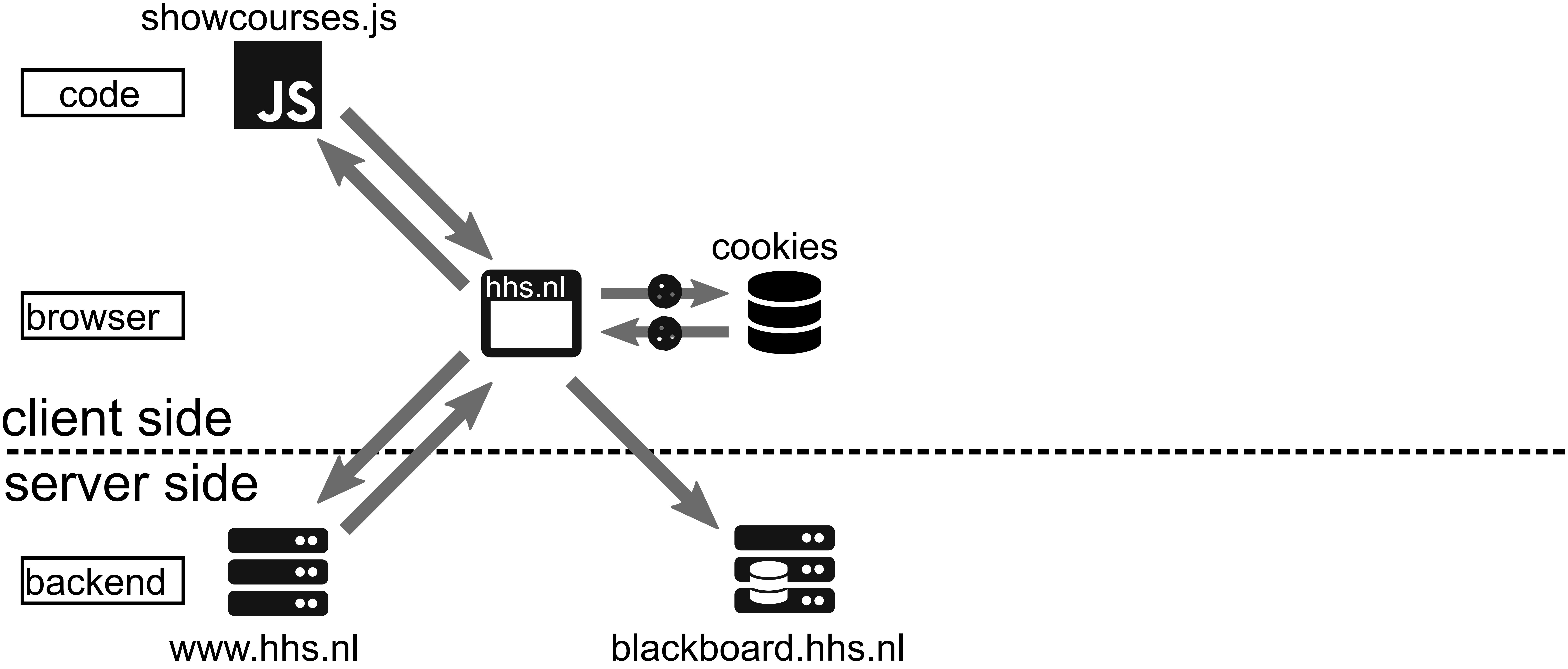
≡

🛡 SAME ORIGIN POLICY (VS CORS)



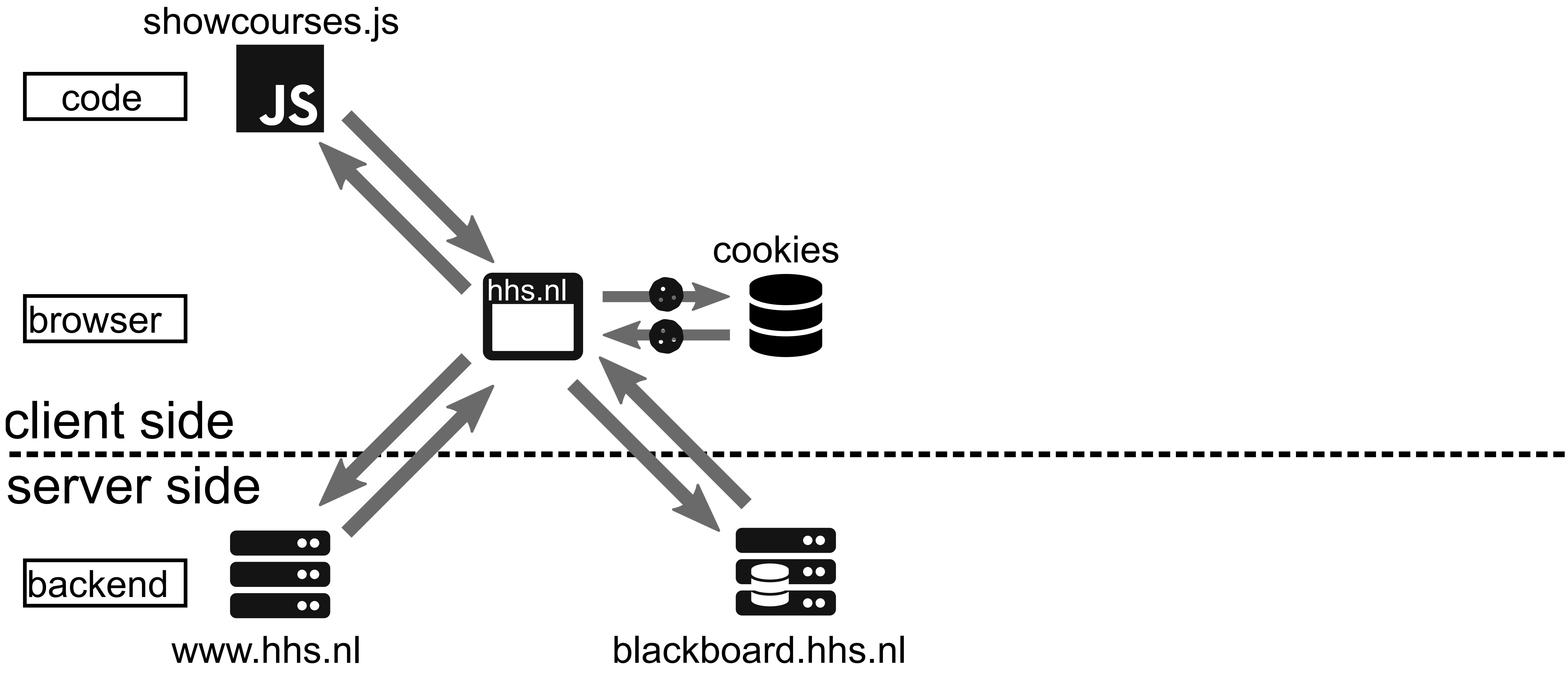
≡

🛡 SAME ORIGIN POLICY (VS CORS)

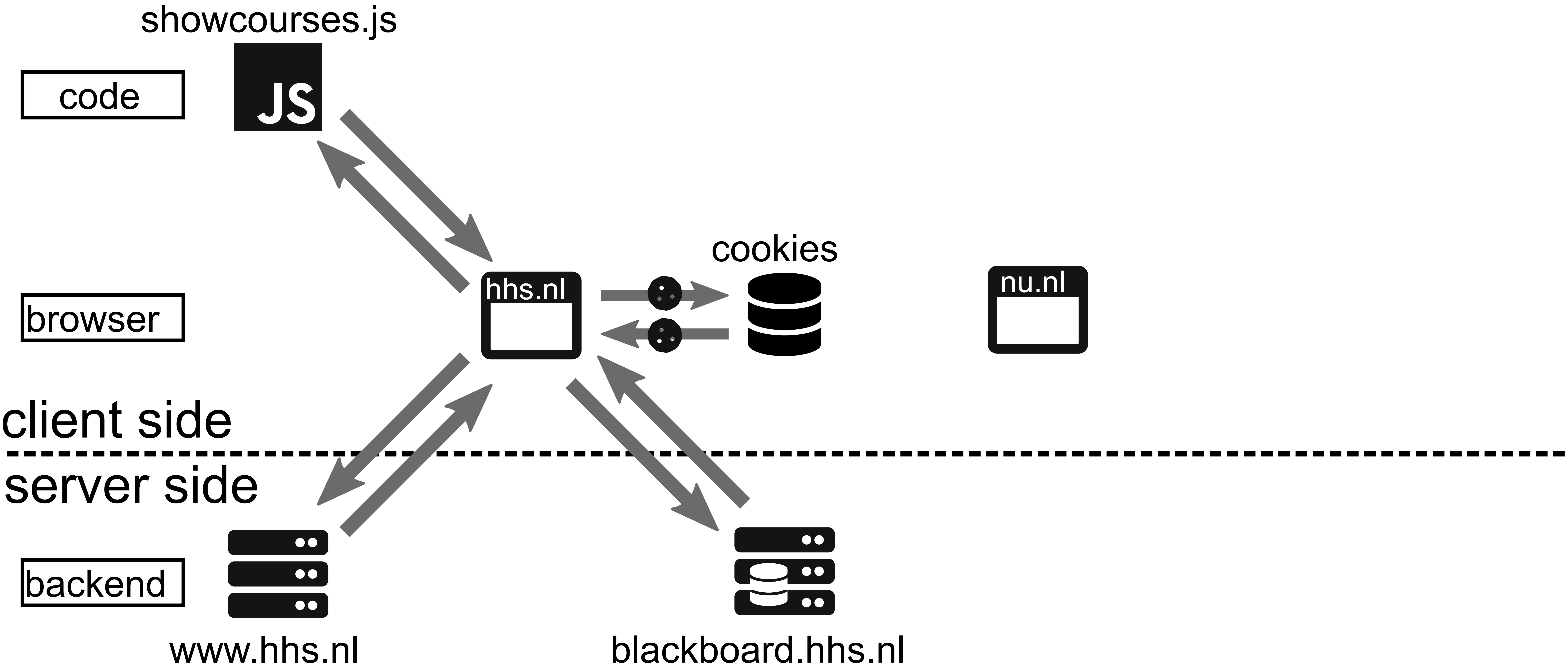


≡

🛡 SAME ORIGIN POLICY (VS CORS)

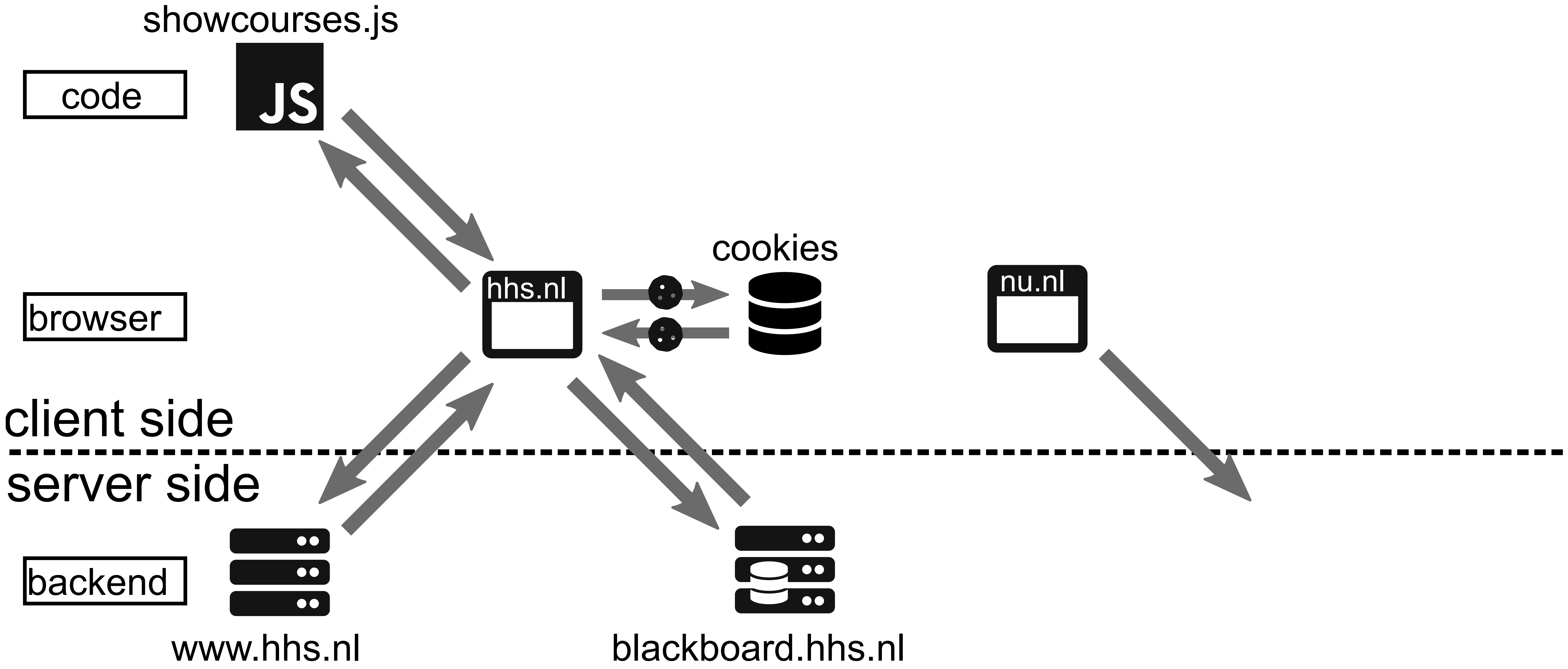


🛡 SAME ORIGIN POLICY (VS CORS)



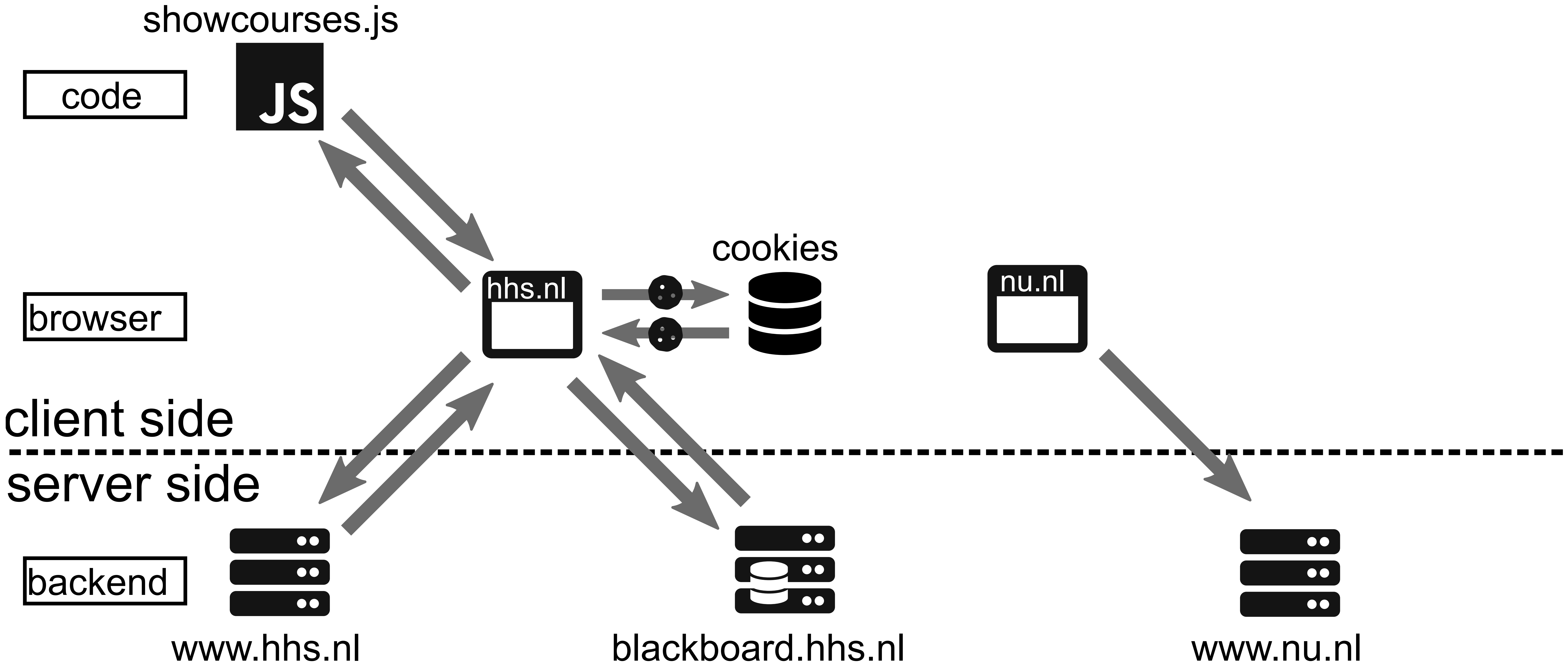
≡

🛡 SAME ORIGIN POLICY (VS CORS)

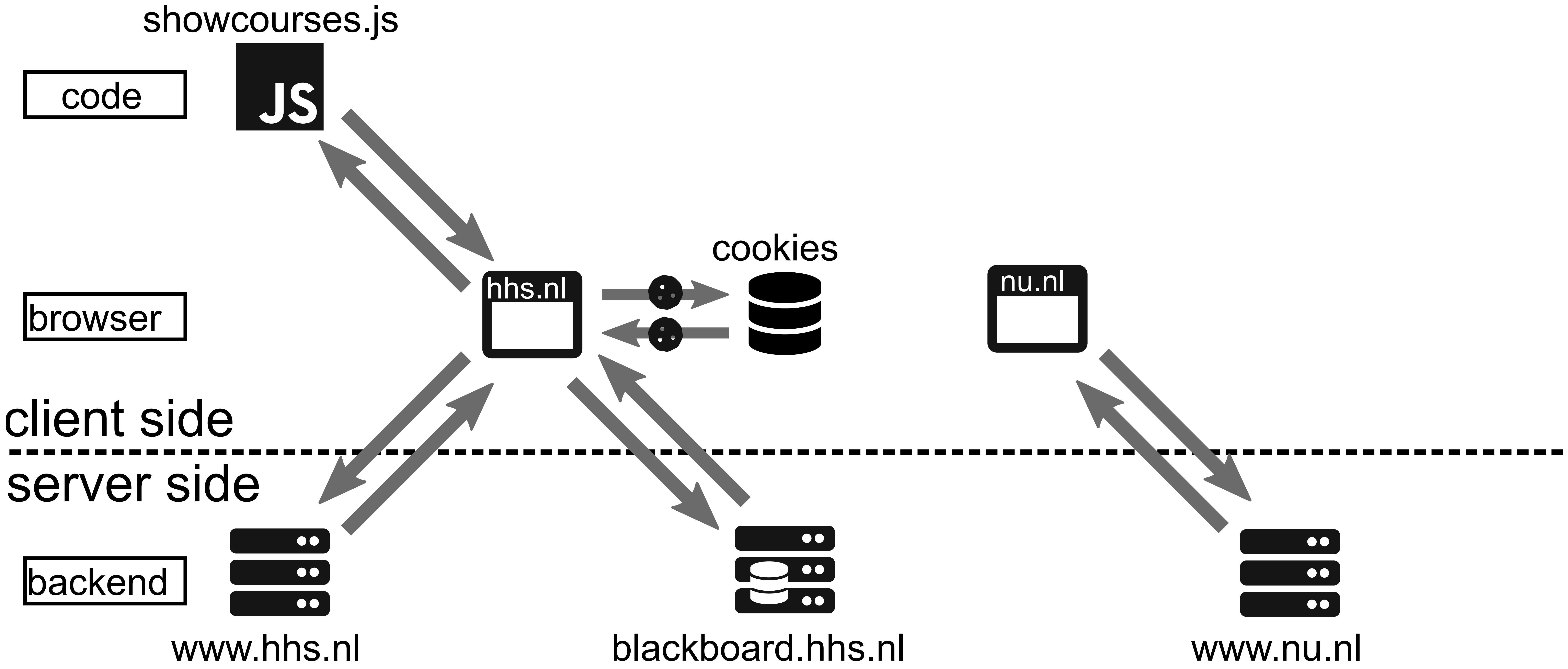


☰

🛡 SAME ORIGIN POLICY (VS CORS)



🛡 SAME ORIGIN POLICY (VS CORS)



showcourses.js

code

browser

hhs.nl

cookies

nu.nl

client side

server side

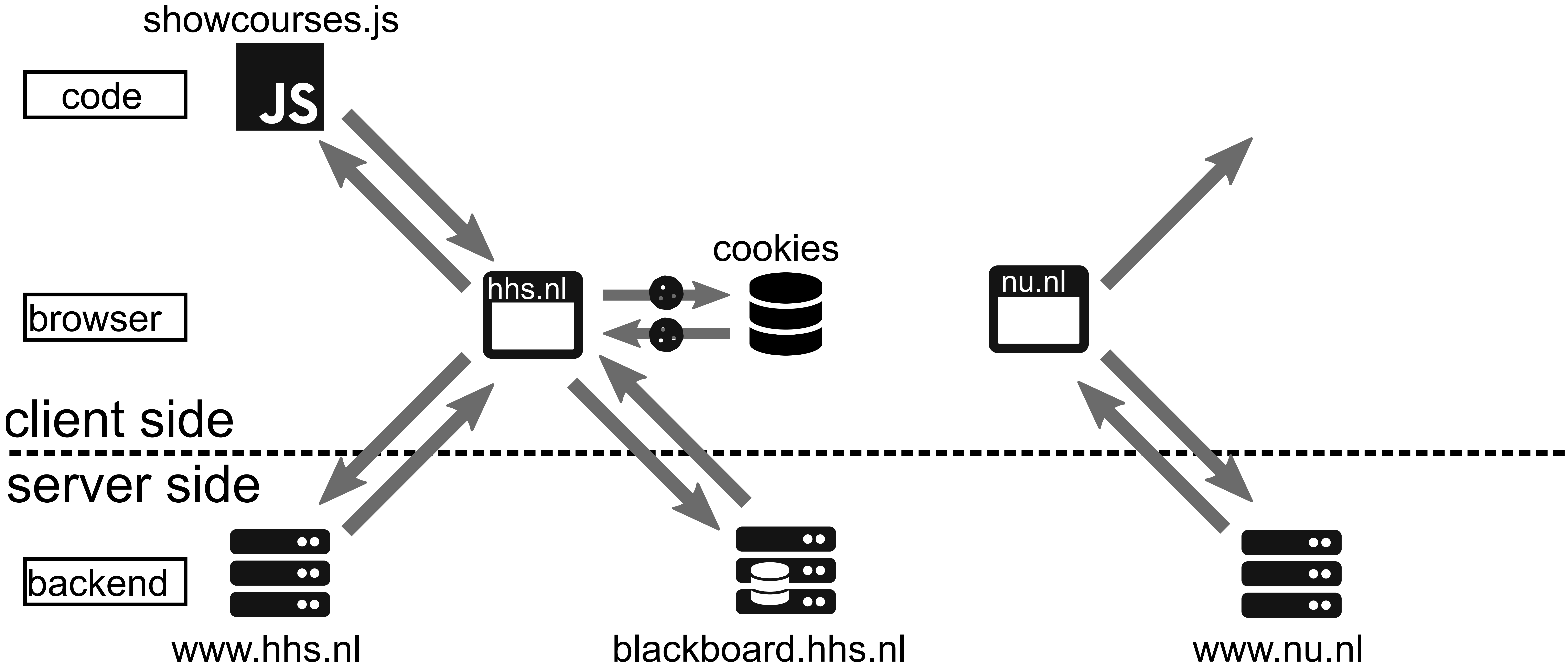
backend

www.hhs.nl

blackboard.hhs.nl

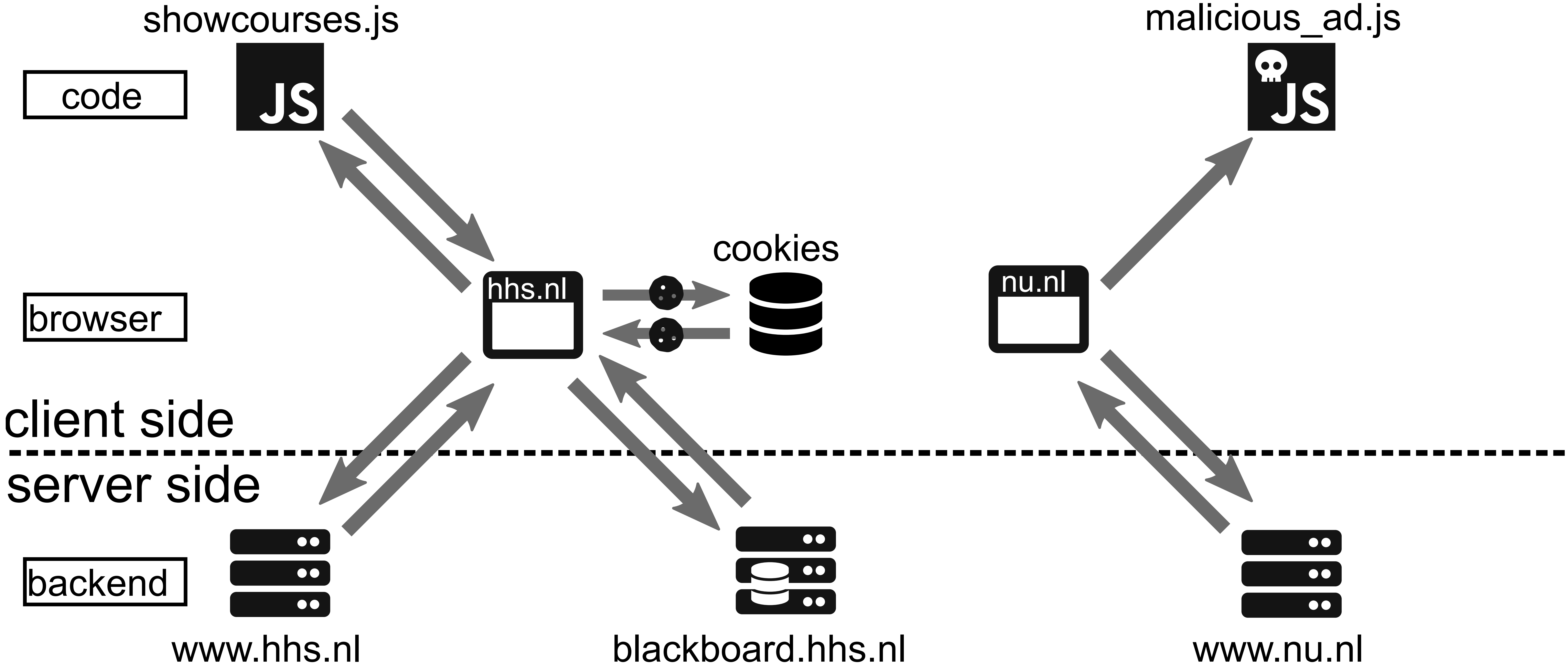
www.nu.nl

🛡 SAME ORIGIN POLICY (VS CORS)



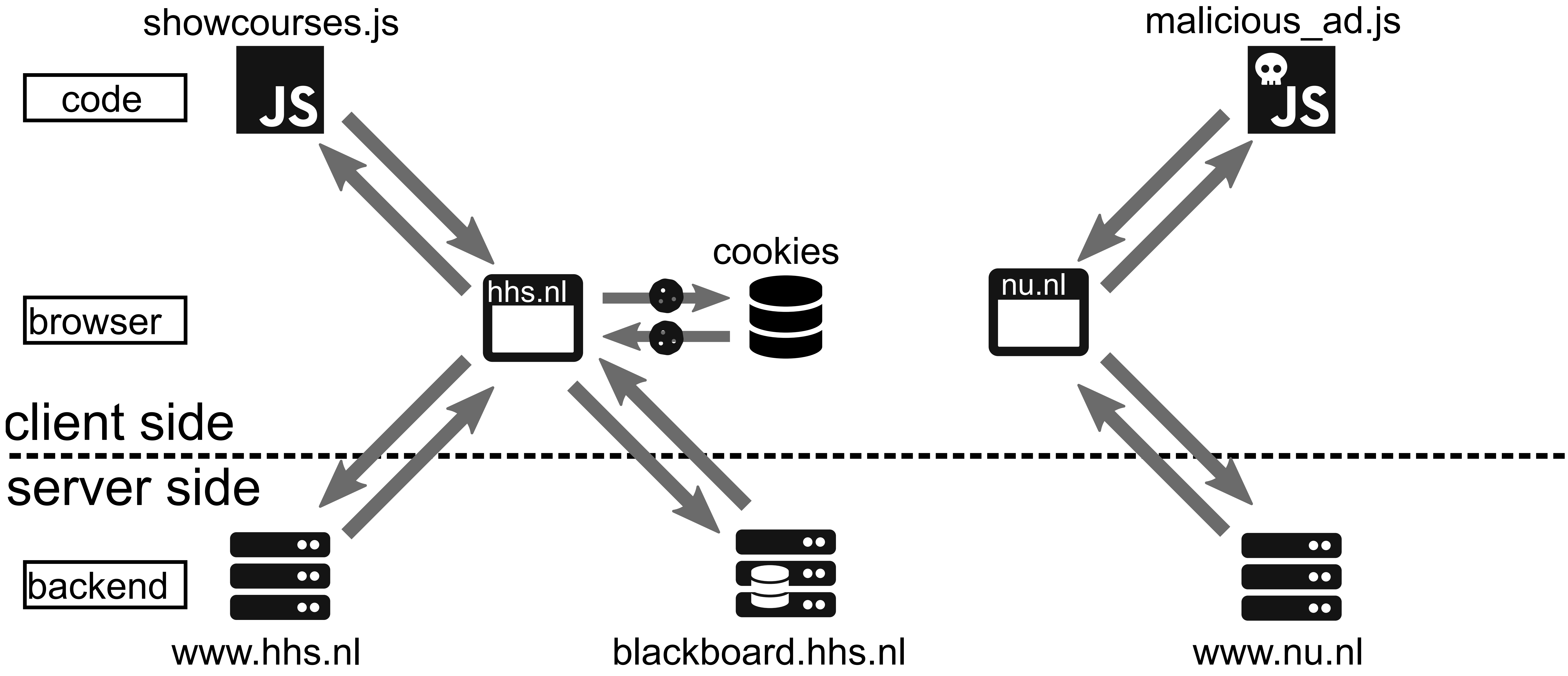
≡

🛡 SAME ORIGIN POLICY (VS CORS)



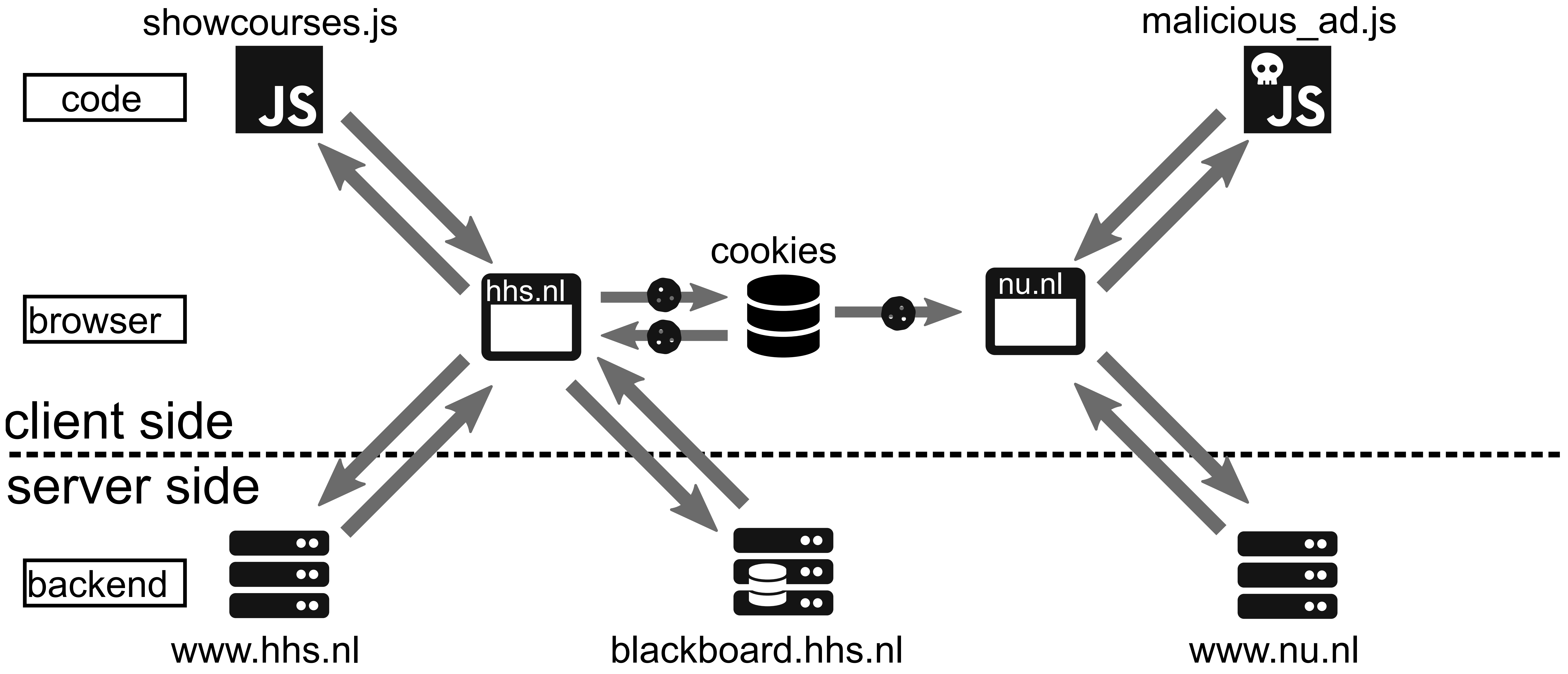
≡

🛡 SAME ORIGIN POLICY (VS CORS)



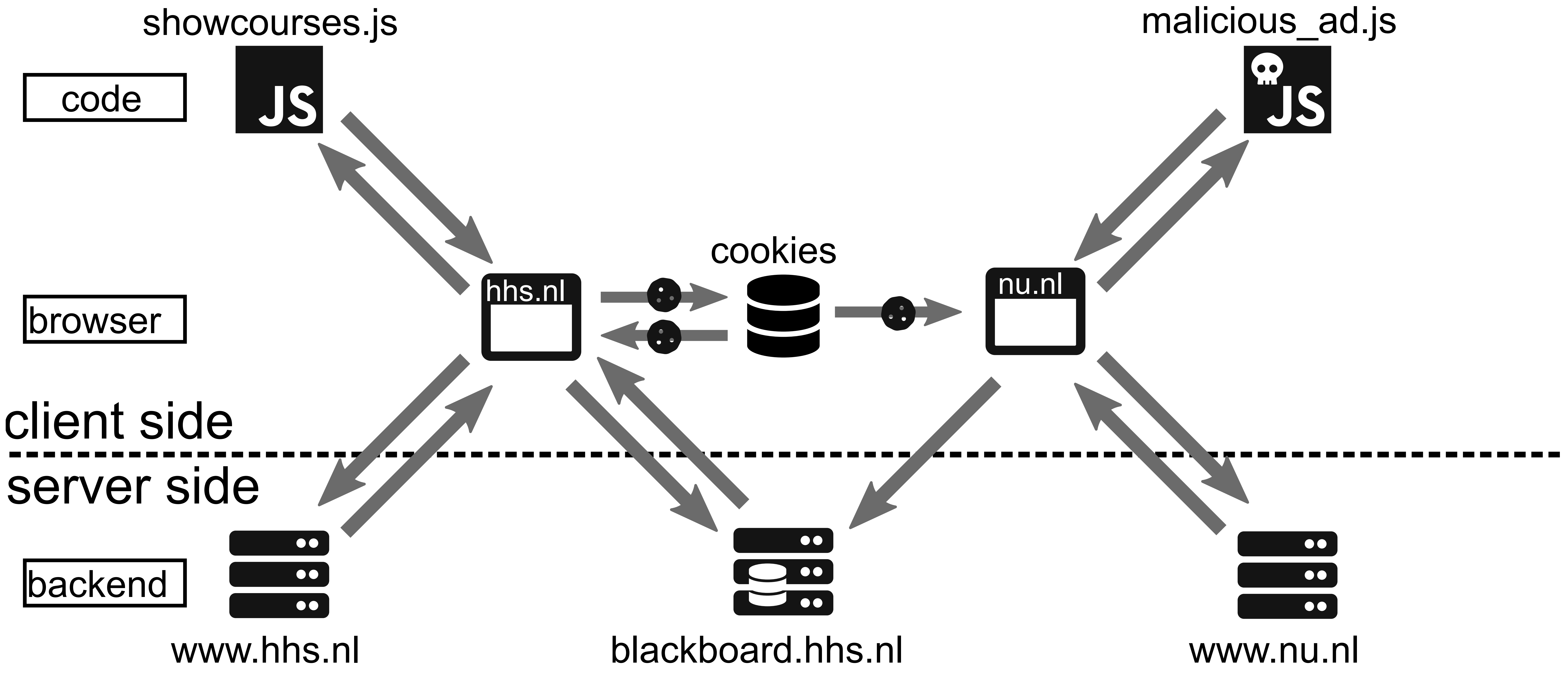
≡

🛡 SAME ORIGIN POLICY (VS CORS)



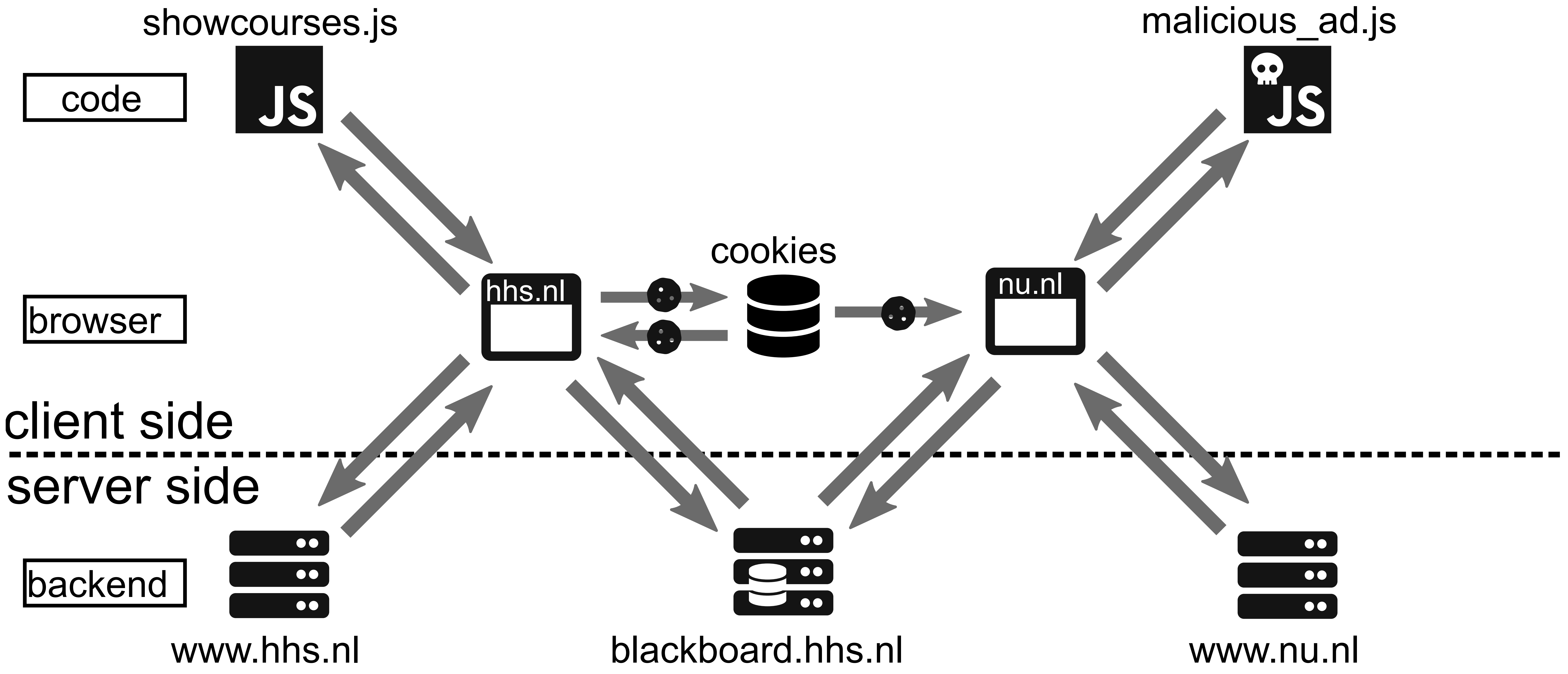
≡

🛡 SAME ORIGIN POLICY (VS CORS)



≡

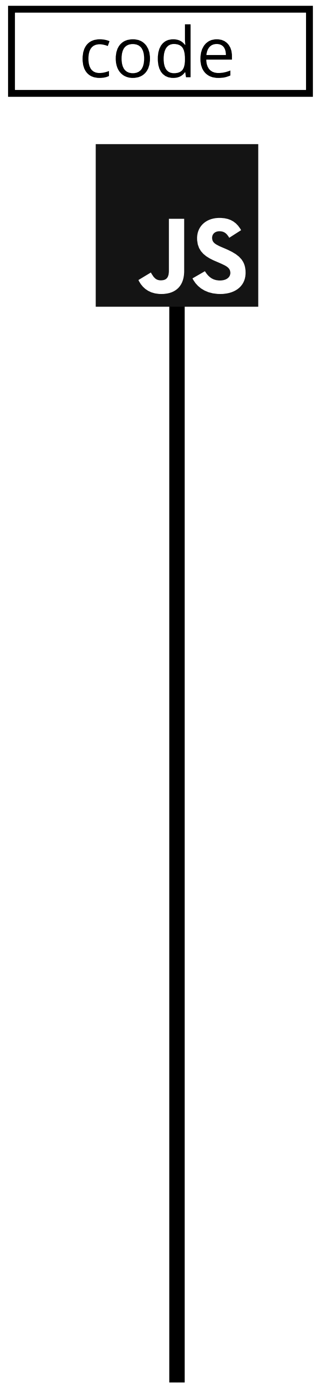
🛡 SAME ORIGIN POLICY (VS CORS)



≡

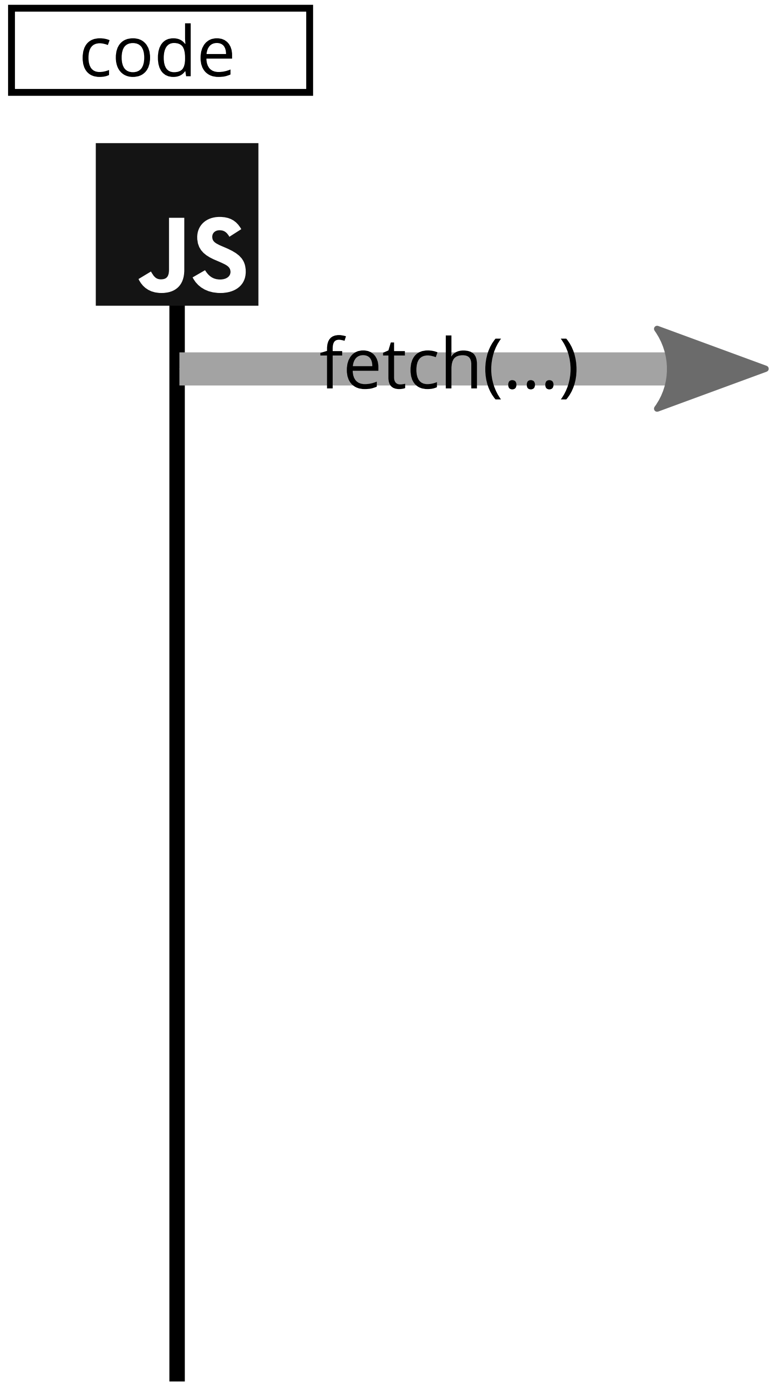
PREFLIGHT

Om te voorkomen dat een GET een onbedoeld effect heeft.



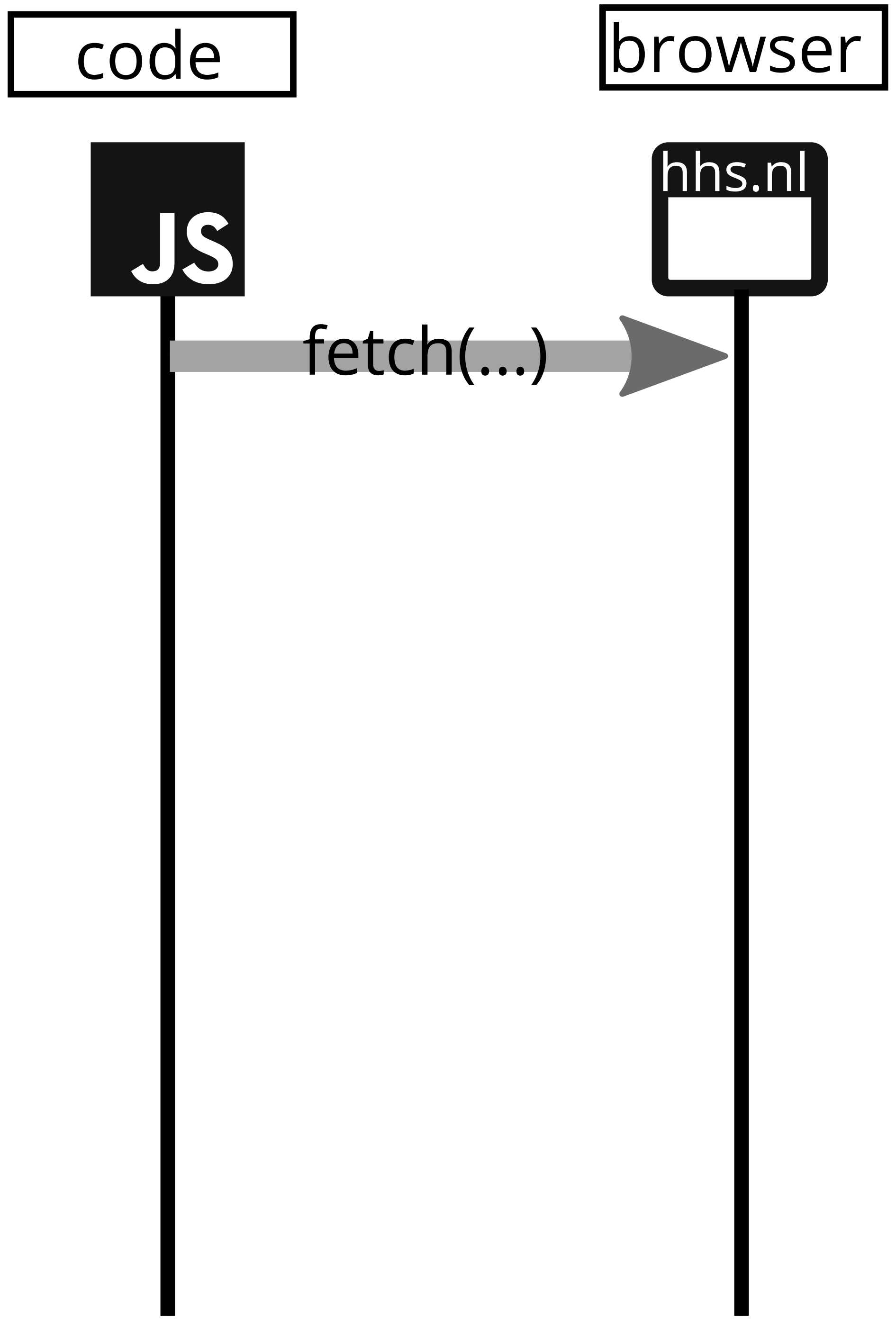
PREFLIGHT

Om te voorkomen dat een GET een onbedoeld effect heeft.



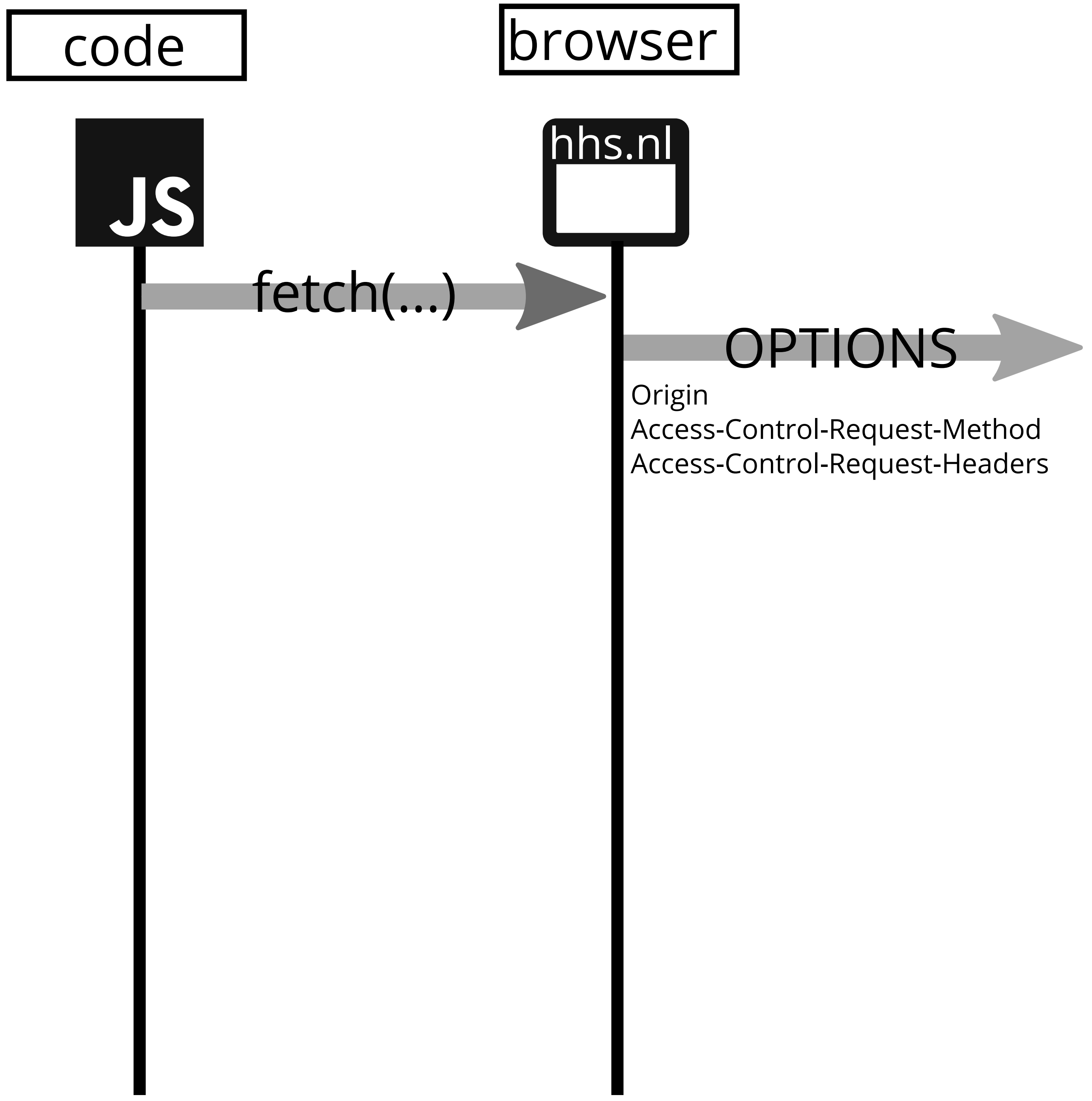
PREFLIGHT

Om te voorkomen dat een GET een onbedoeld effect heeft.



PREFLIGHT

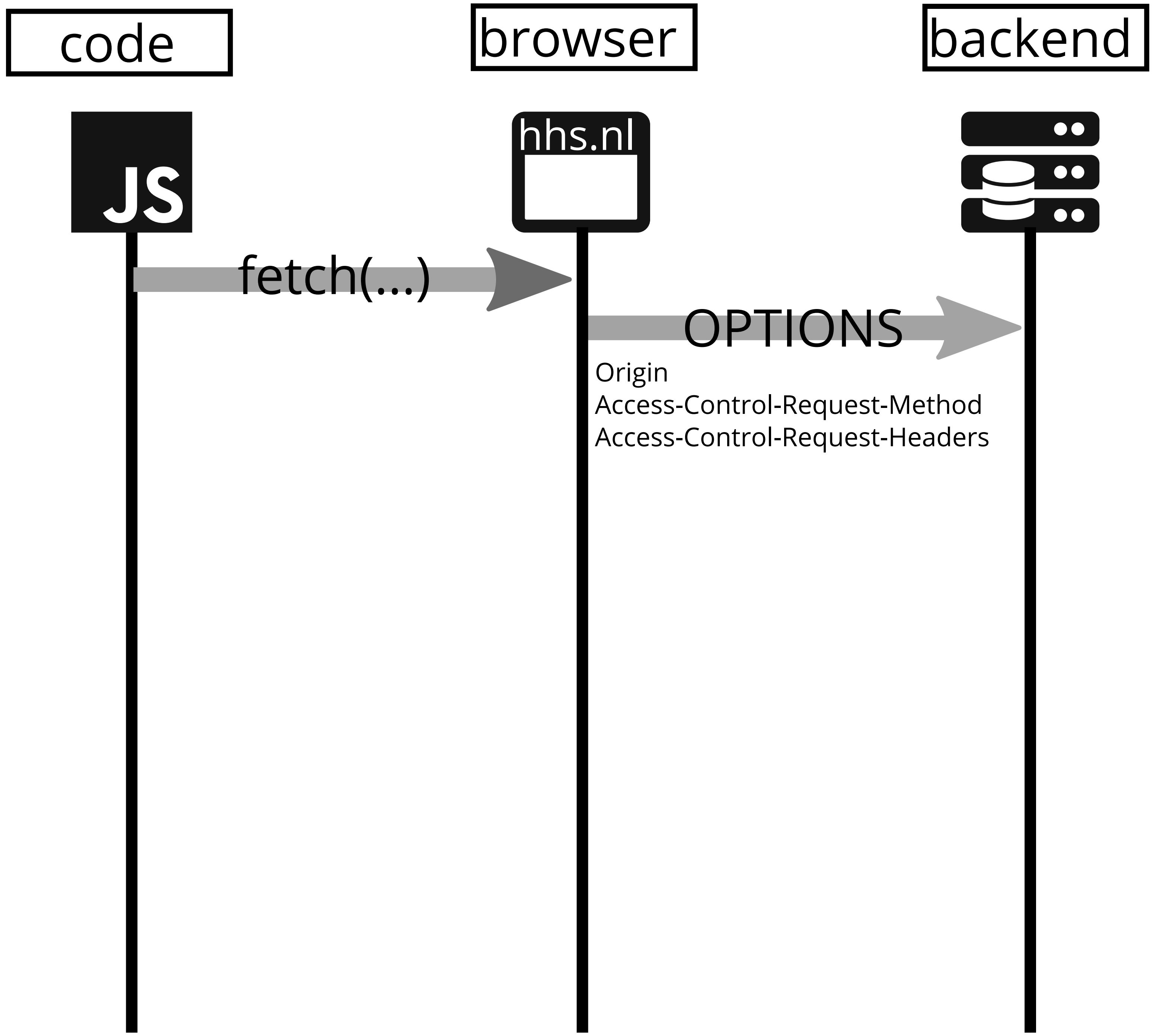
Om te voorkomen dat een GET een onbedoeld effect heeft.



≡

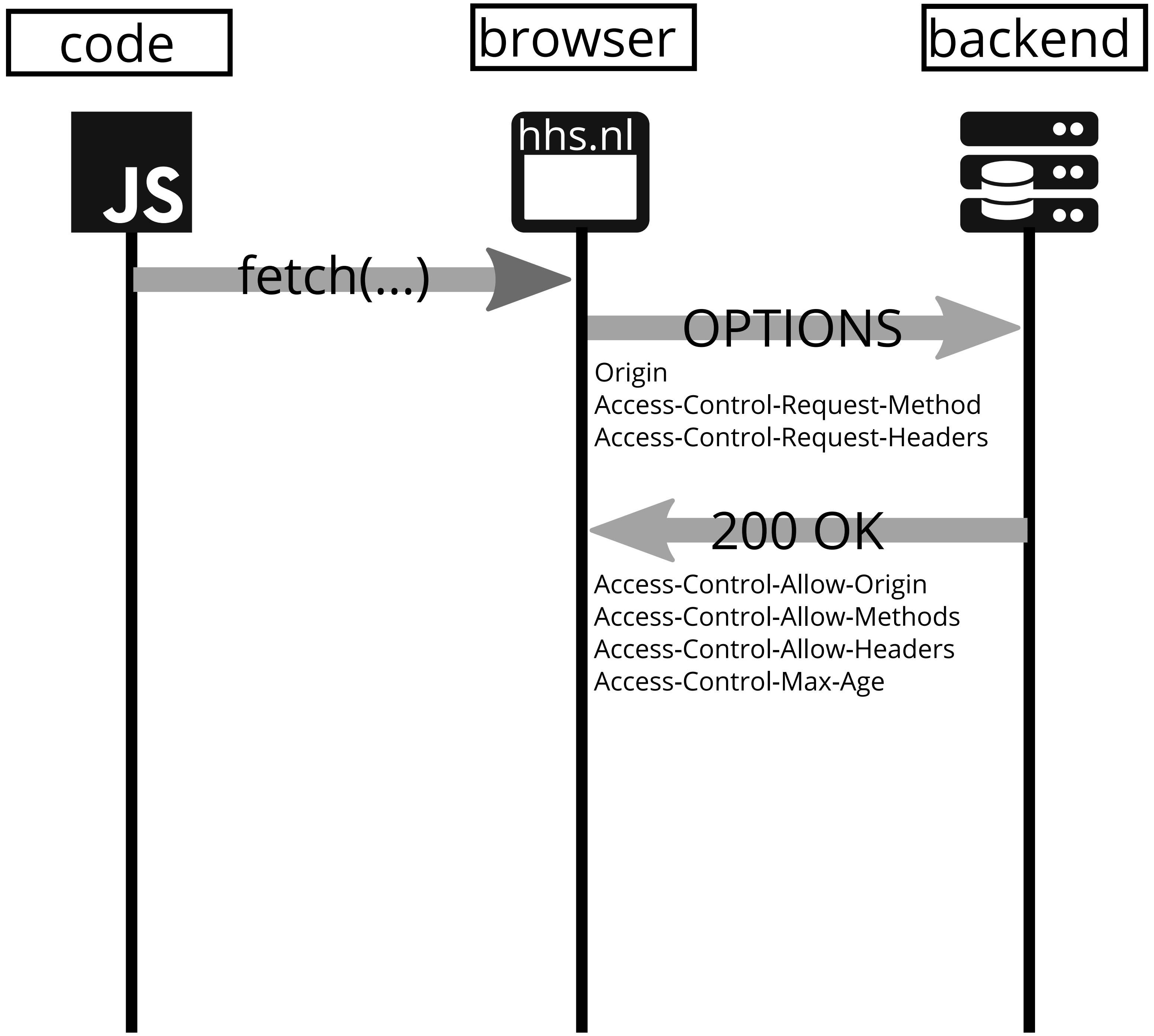
PREFLIGHT

Om te voorkomen dat een GET een onbedoeld effect heeft.



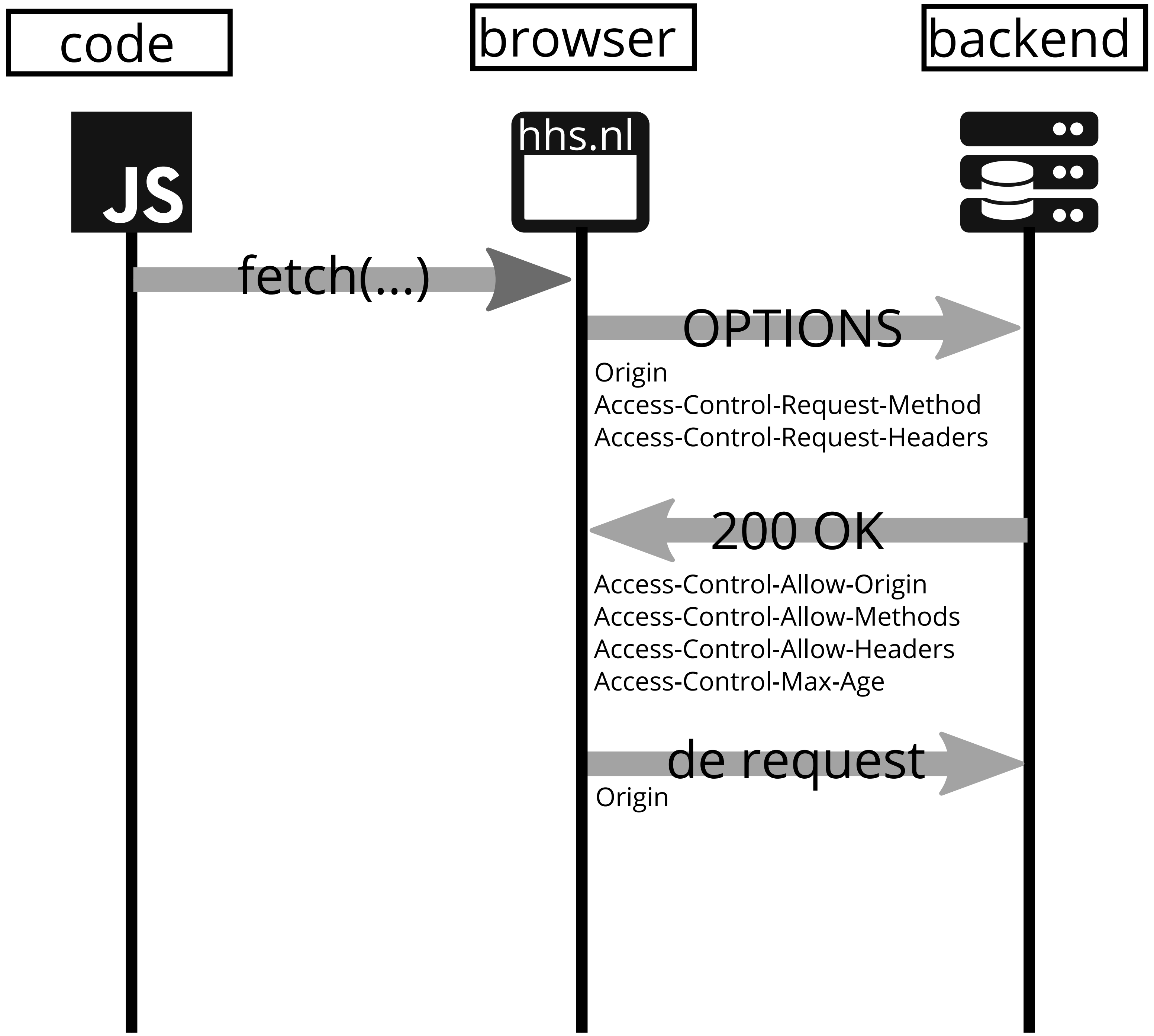
PREFLIGHT

Om te voorkomen dat een GET een onbedoeld effect heeft.


code
browser
backend
JS
hhs.nl
fetch(...)
OPTIONS
Origin
Access-Control-Request-Method
Access-Control-Request-Headers
200 OK
Access-Control-Allow-Origin
Access-Control-Allow-Methods
Access-Control-Allow-Headers
Access-Control-Max-Age

PREFLIGHT

Om te voorkomen dat een GET een onbedoeld effect heeft.



fetch(...)

OPTIONS

Origin
Access-Control-Request-Method
Access-Control-Request-Headers

200 OK

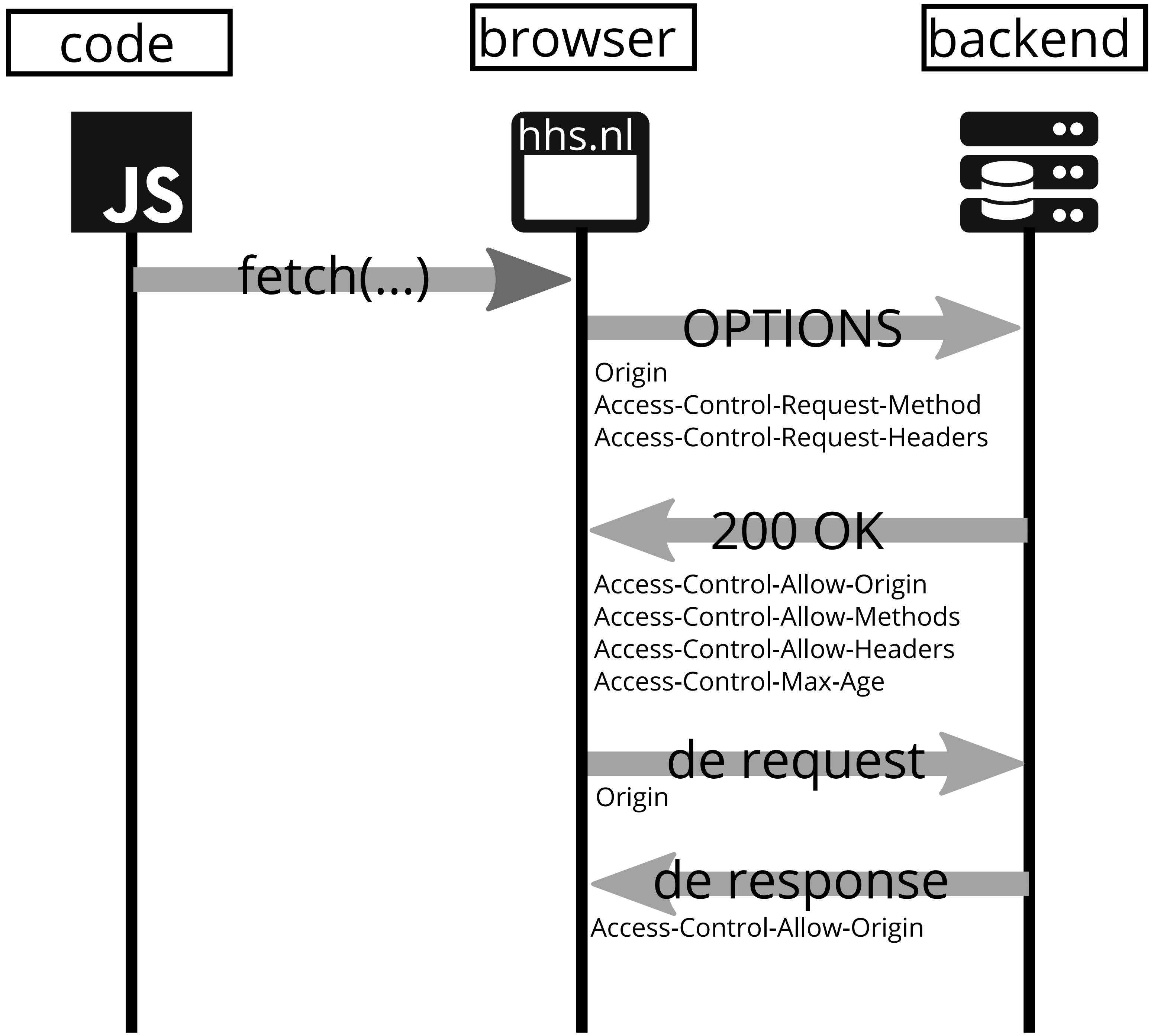
Access-Control-Allow-Origin
Access-Control-Allow-Methods
Access-Control-Allow-Headers
Access-Control-Max-Age

de request

Origin

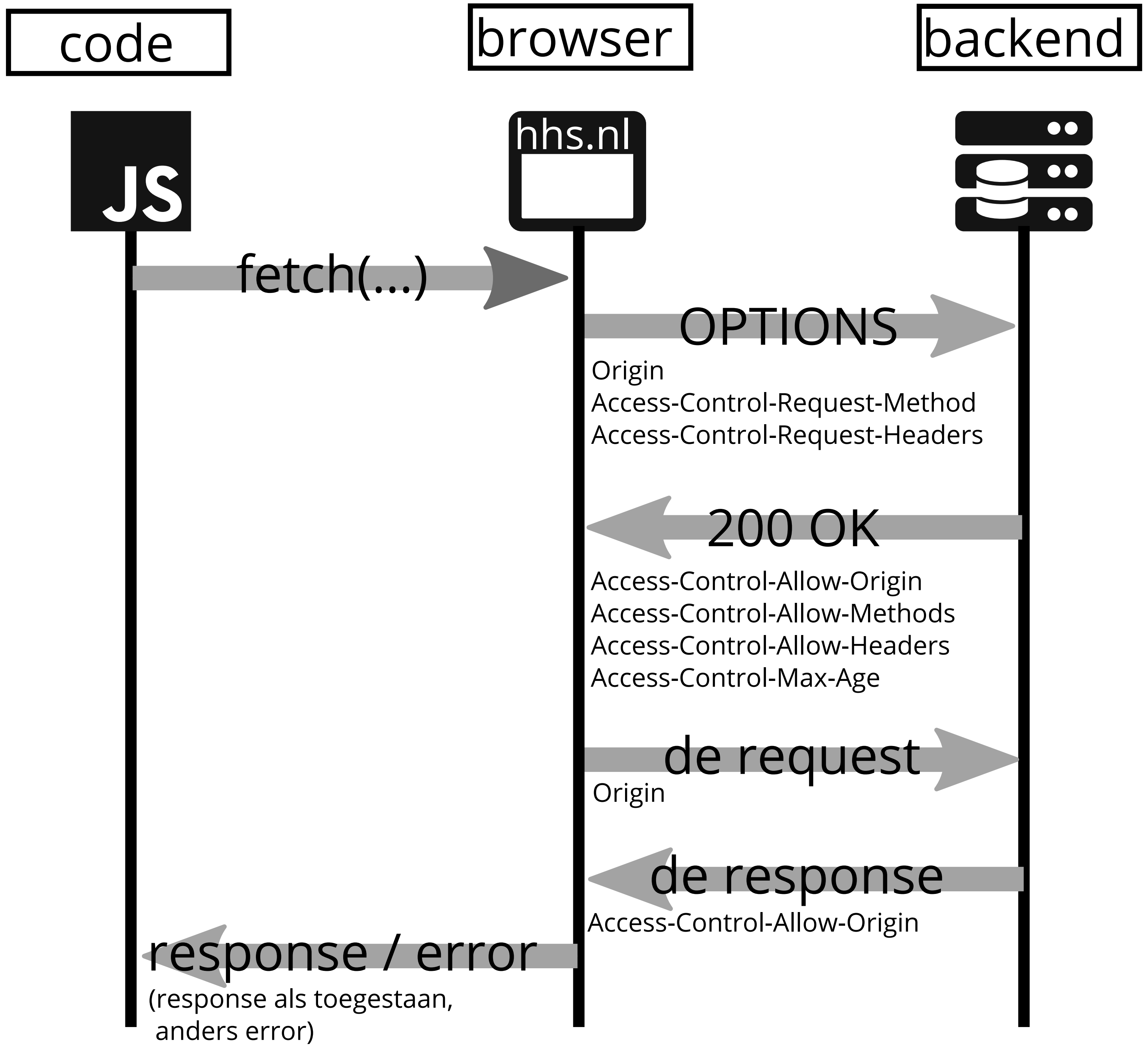
PREFLIGHT

Om te voorkomen dat een GET een onbedoeld effect heeft.


code
browser
backend
JS
hhs.nl
fetch(...)
OPTIONS
Origin
Access-Control-Request-Method
Access-Control-Request-Headers
200 OK
Access-Control-Allow-Origin
Access-Control-Allow-Methods
Access-Control-Allow-Headers
Access-Control-Max-Age
de request
Origin
de response
Access-Control-Allow-Origin

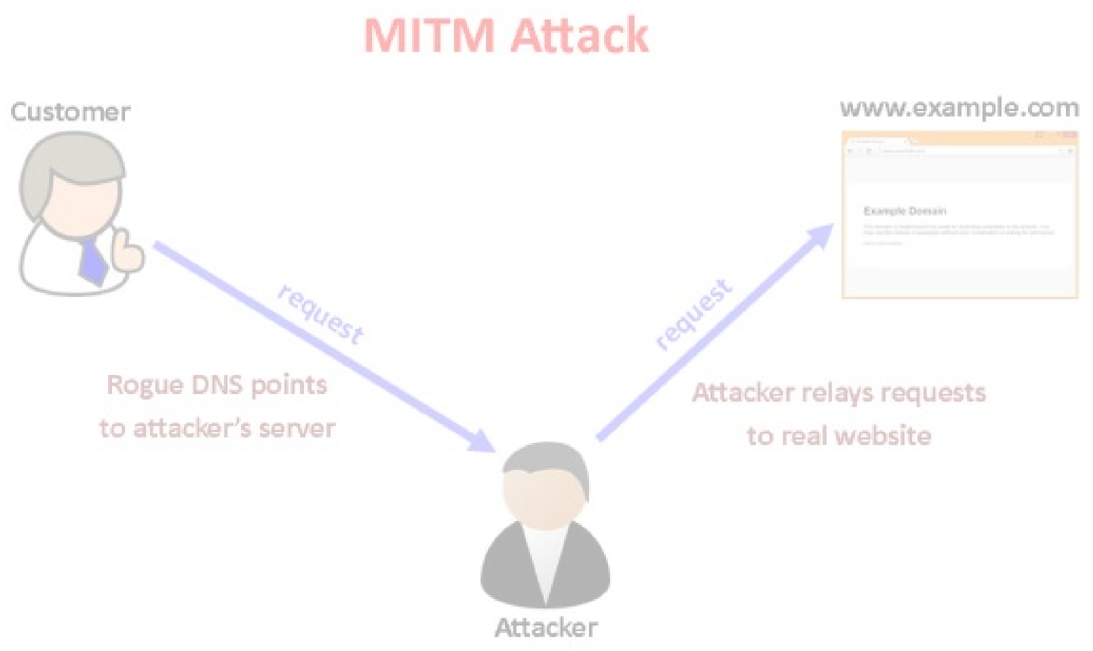
PREFLIGHT

Om te voorkomen dat een GET een onbedoeld effect heeft.


code
browser
backend
JS
hhs.nl
fetch(...)
OPTIONS
Origin
Access-Control-Request-Method
Access-Control-Request-Headers
200 OK
Access-Control-Allow-Origin
Access-Control-Allow-Methods
Access-Control-Allow-Headers
Access-Control-Max-Age
de request
Origin
de response
Access-Control-Allow-Origin
response / error
(response als toegestaan,
anders error)

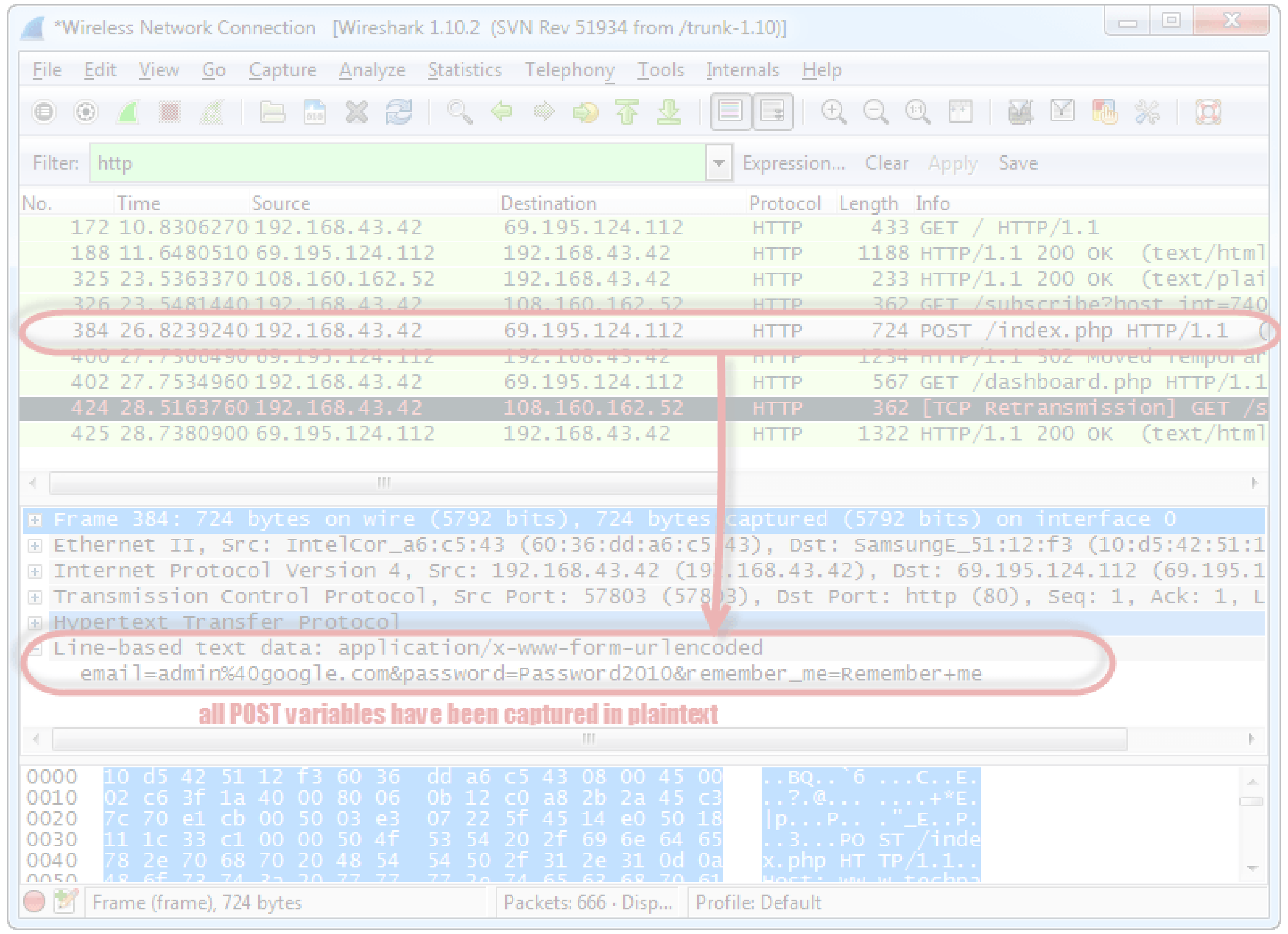
TWEE PROBLEMEN MET HTTP

Man in the middle attack



- Certificaat nodig om te bewijzen dat jij de échte server bent.
- Voorbeeld: Russische aanval vanuit een auto buiten het OPCW gebouw in Scheveningen

Stiekem meeluisteren


*Wireless Network Connection [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

Filter: http

```
No.   Time        Source          Destination      Protocol Length Info
  172 10.8306270 192.168.43.42   69.195.124.112   HTTP      433 GET / HTTP/1.1
  188 11.6480510 69.195.124.112  192.168.43.42    HTTP     1188 HTTP/1.1 200 OK  (text/html
  325 23.5363370 108.160.162.52  192.168.43.42    HTTP      233 HTTP/1.1 200 OK  (text/plai
  326 23.5481440 192.168.43.42   108.160.162.52   HTTP      362 GET /subscribe?host_int=740
  384 26.8239240 192.168.43.42   69.195.124.112   HTTP      724 POST /index.php HTTP/1.1
  400 27.7300490 69.195.124.112  192.168.43.42    HTTP     1234 HTTP/1.1 302 Moved Temporar
  402 27.7534960 192.168.43.42   69.195.124.112   HTTP      567 GET /dashboard.php HTTP/1.1
  424 28.5163760 192.168.43.42   108.160.162.52   HTTP      362 [TCP Retransmission] GET /s
  425 28.7380900 69.195.124.112  192.168.43.42    HTTP     1322 HTTP/1.1 200 OK  (text/html
```

```
⊞ Frame 384: 724 bytes on wire (5792 bits), 724 bytes captured (5792 bits) on interface 0
⊞ Ethernet II, Src: IntelCor_a6:c5:43 (60:36:dd:a6:c5:43), Dst: SamsungE_51:12:f3 (10:d5:42:51:1
⊞ Internet Protocol Version 4, Src: 192.168.43.42 (192.168.43.42), Dst: 69.195.124.112 (69.195.1
⊞ Transmission Control Protocol, Src Port: 57803 (57803), Dst Port: http (80), Seq: 1, Ack: 1, L
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: application/x-www-form-urlencoded
    email=admin%40google.com&password=Password2010&remember_me=Remember+me
```

all POST variables have been captured in plaintext

```
0000  10 d5 42 51 12 f3 60 36  dd a6 c5 43 08 00 45 00   ..BQ..`6 ...C..E.
0010  02 c6 3f 1a 40 00 80 06  0b 12 c0 a8 2b 2a 45 c3   ..?.@... ....+*E.
0020  7c 70 e1 cb 00 50 03 e3  07 22 5f 45 14 e0 50 18   |p...P.. ."_E..P.
0030  11 1c 33 c1 00 00 50 4f  53 54 20 2f 69 6e 64 65   ..3...PO ST /inde
0040  78 2e 70 68 70 20 48 54  54 50 2f 31 2e 31 0d 0a   x.php HT TP/1.1..
```

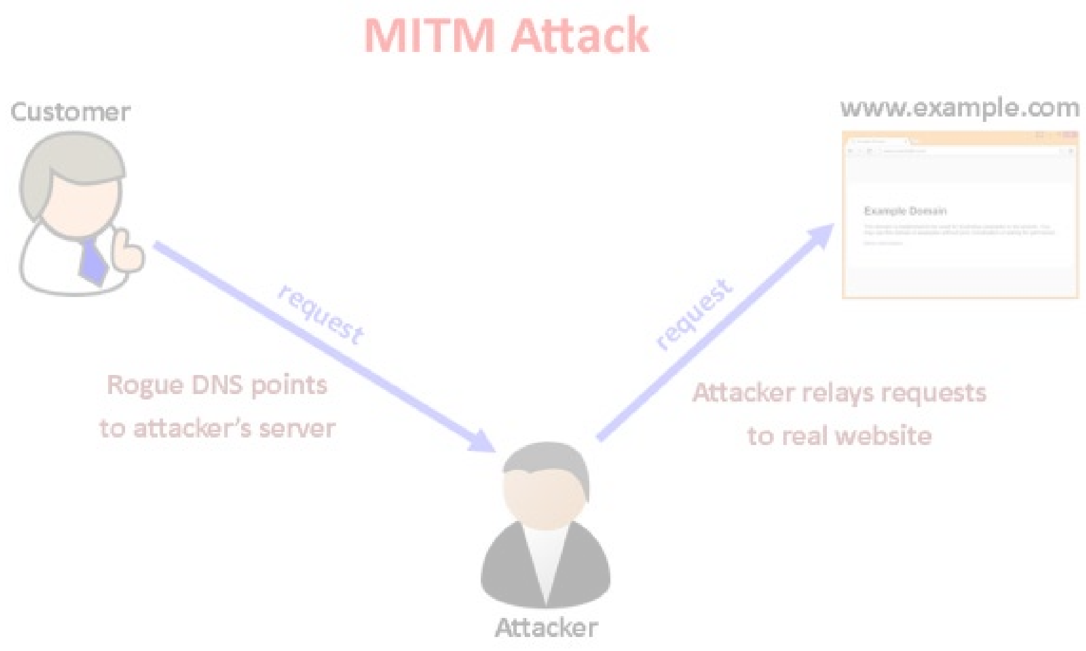
Frame (frame), 724 bytes | Packets: 666 · Disp... | Profile: Default

(bron: guru99)

- Zonder Wifi-beveiliging kan iedereen alle HTTP requests zien

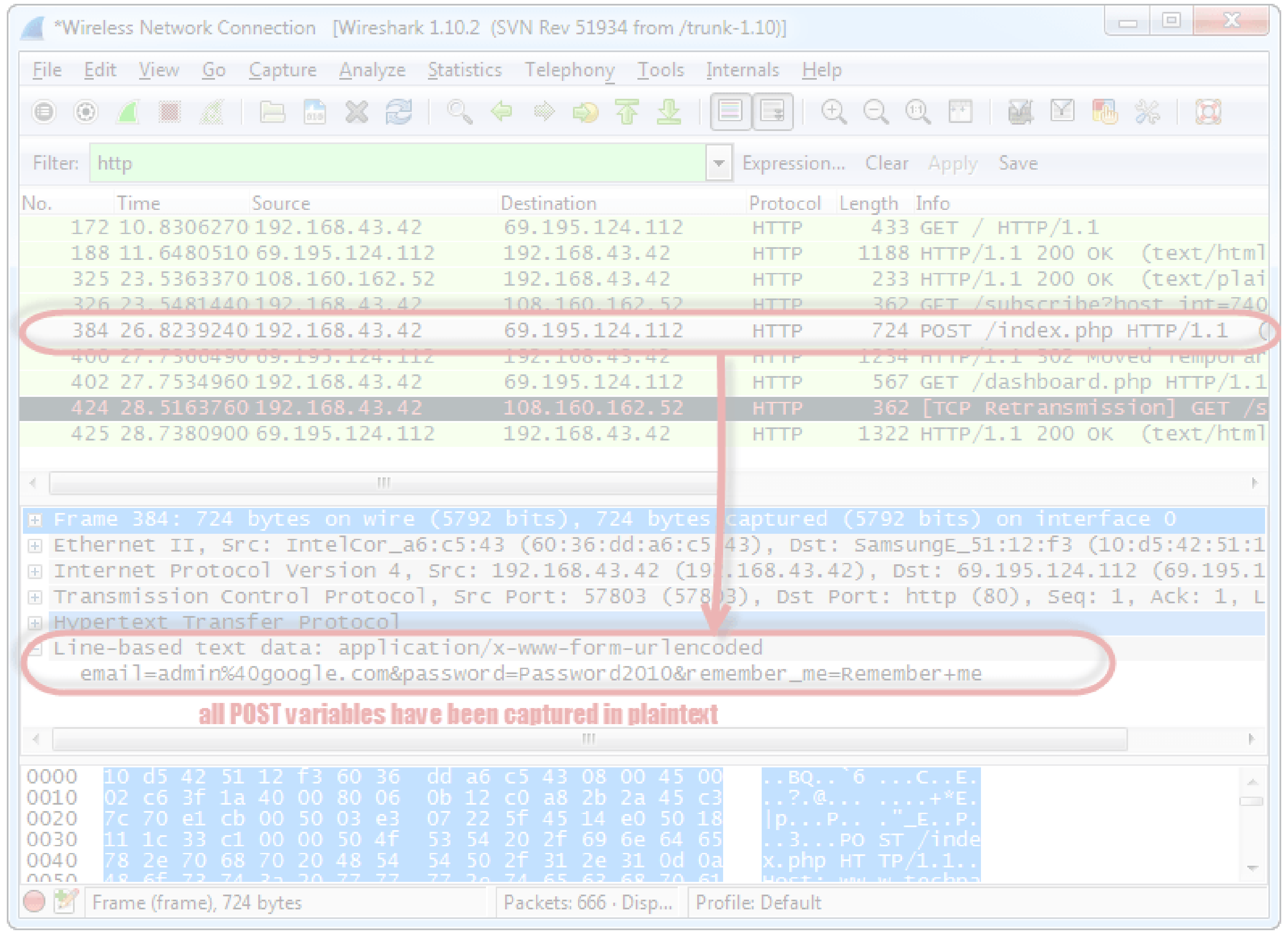
TWEE PROBLEMEN MET HTTP

Man in the middle attack



- Certificaat nodig om te bewijzen dat jij de échte server bent.
- Voorbeeld: Russische aanval vanuit een auto buiten het OPCW gebouw in Scheveningen

Stiekem meeluisteren


*Wireless Network Connection [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: http
Expression... Clear Apply Save

```
No.   Time        Source          Destination     Protocol Length Info
  172 10.8306270 192.168.43.42   69.195.124.112  HTTP      433 GET / HTTP/1.1
  188 11.6480510 69.195.124.112  192.168.43.42   HTTP     1188 HTTP/1.1 200 OK  (text/html
  325 23.5363370 108.160.162.52  192.168.43.42   HTTP      233 HTTP/1.1 200 OK  (text/plai
  326 23.5481440 192.168.43.42   108.160.162.52  HTTP      362 GET /subscribe?host_int=740
  384 26.8239240 192.168.43.42   69.195.124.112  HTTP      724 POST /index.php HTTP/1.1  (
  400 27.7506490 69.195.124.112  192.168.43.42   HTTP     1234 HTTP/1.1 302 Moved Temporar
  402 27.7534960 192.168.43.42   69.195.124.112  HTTP      567 GET /dashboard.php HTTP/1.1
  424 28.5163760 192.168.43.42   108.160.162.52  HTTP      362 [TCP Retransmission] GET /s
  425 28.7380900 69.195.124.112  192.168.43.42   HTTP     1322 HTTP/1.1 200 OK  (text/html

⊞ Frame 384: 724 bytes on wire (5792 bits), 724 bytes captured (5792 bits) on interface 0
⊞ Ethernet II, Src: IntelCor_a6:c5:43 (60:36:dd:a6:c5:43), Dst: SamsungE_51:12:f3 (10:d5:42:51:1
⊞ Internet Protocol Version 4, Src: 192.168.43.42 (192.168.43.42), Dst: 69.195.124.112 (69.195.1
⊞ Transmission Control Protocol, Src Port: 57803 (57803), Dst Port: http (80), Seq: 1, Ack: 1, L
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: application/x-www-form-urlencoded
    email=admin%40google.com&password=Password2010&remember_me=Remember+me

0000  10 d5 42 51 12 f3 60 36  dd a6 c5 43 08 00 45 00   ..BQ..`6 ...C..E.
0010  02 c6 3f 1a 40 00 80 06  0b 12 c0 a8 2b 2a 45 c3   ..?.@... ....+*E.
0020  7c 70 e1 cb 00 50 03 e3  07 22 5f 45 14 e0 50 18   |p...P.. ."_E..P.
0030  11 1c 33 c1 00 00 50 4f  53 54 20 2f 69 6e 64 65   ..3...PO ST /inde
0040  78 2e 70 68 70 20 48 54  54 50 2f 31 2e 31 0d 0a   x.php HT TP/1.1..
```

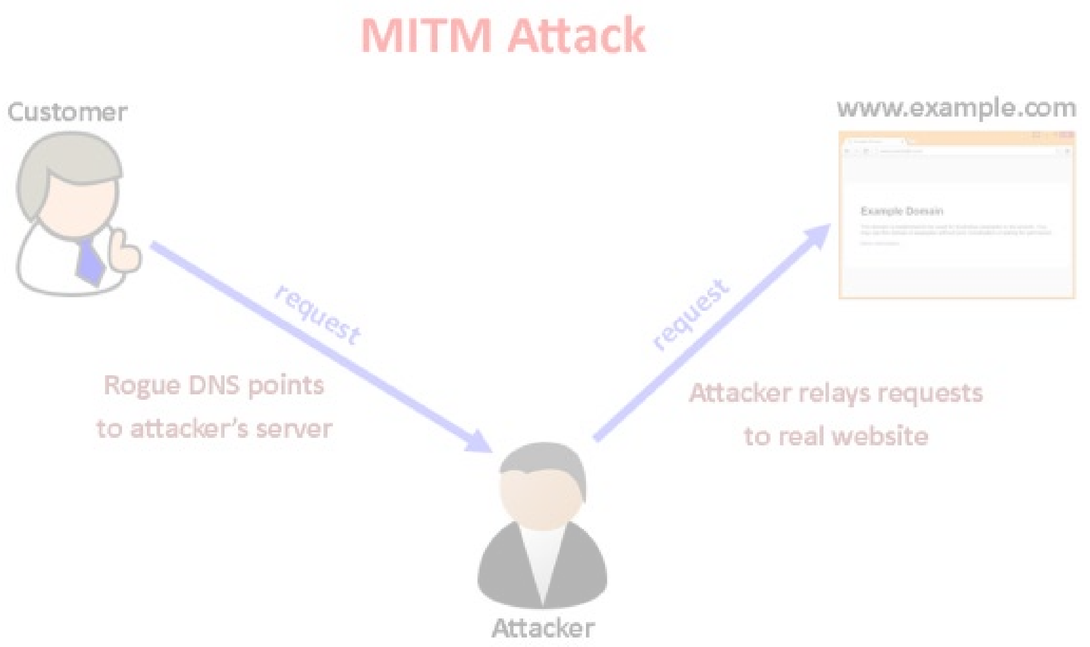
all POST variables have been captured in plaintext

(bron: guru99)

- Zonder Wifi-beveiliging kan iedereen alle HTTP requests zien

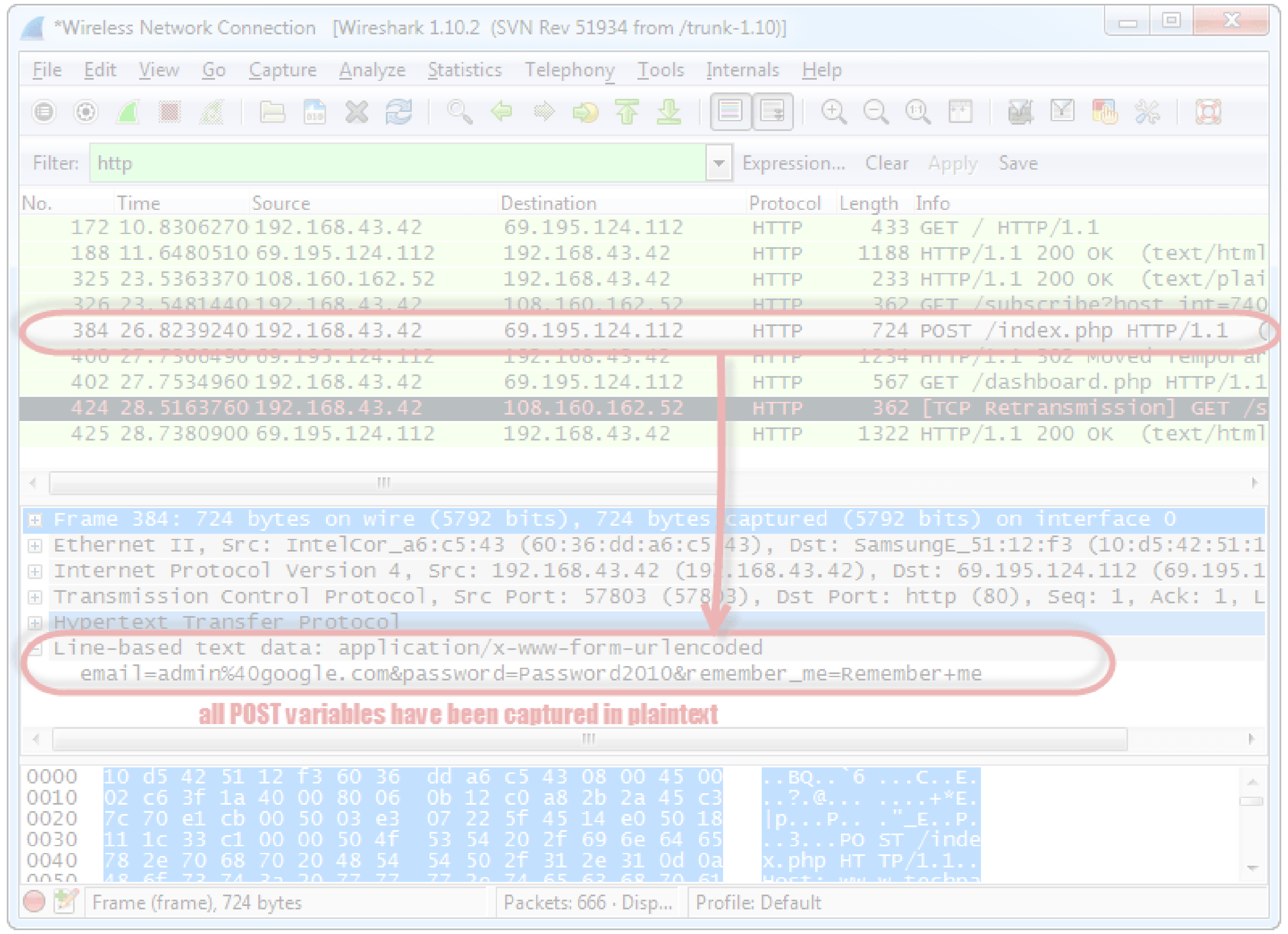
TWEE PROBLEMEN MET HTTP

Man in the middle attack



- Certificaat nodig om te bewijzen dat jij de échte server bent.
- Voorbeeld: Russische aanval vanuit een auto buiten het OPCW gebouw in Scheveningen

Stiekem meeluisteren


*Wireless Network Connection [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

Filter: http

```
No.   Time       Source          Destination      Protocol Length Info
  172 10.8306270 192.168.43.42   69.195.124.112   HTTP      433 GET / HTTP/1.1
  188 11.6480510 69.195.124.112  192.168.43.42    HTTP     1188 HTTP/1.1 200 OK  (text/html
  325 23.5363370 108.160.162.52  192.168.43.42    HTTP      233 HTTP/1.1 200 OK  (text/plai
  326 23.5481440 192.168.43.42   108.160.162.52   HTTP      362 GET /subscribe?host_int=740
  384 26.8239240 192.168.43.42   69.195.124.112   HTTP      724 POST /index.php HTTP/1.1
  400 27.7500490 69.195.124.112  192.168.43.42    HTTP     1234 HTTP/1.1 302 Moved Temporar
  402 27.7534960 192.168.43.42   69.195.124.112   HTTP      567 GET /dashboard.php HTTP/1.1
  424 28.5163760 192.168.43.42   108.160.162.52   HTTP      362 [TCP Retransmission] GET /s
  425 28.7380900 69.195.124.112  192.168.43.42    HTTP     1322 HTTP/1.1 200 OK  (text/html
```

```
⊞ Frame 384: 724 bytes on wire (5792 bits), 724 bytes captured (5792 bits) on interface 0
⊞ Ethernet II, Src: IntelCor_a6:c5:43 (60:36:dd:a6:c5:43), Dst: SamsungE_51:12:f3 (10:d5:42:51:1
⊞ Internet Protocol Version 4, Src: 192.168.43.42 (192.168.43.42), Dst: 69.195.124.112 (69.195.1
⊞ Transmission Control Protocol, Src Port: 57803 (57803), Dst Port: http (80), Seq: 1, Ack: 1, L
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: application/x-www-form-urlencoded
    email=admin%40google.com&password=Password2010&remember_me=Remember+me
```

all POST variables have been captured in plaintext

```
0000  10 d5 42 51 12 f3 60 36  dd a6 c5 43 08 00 45 00   ..BQ..`6 ...C..E.
0010  02 c6 3f 1a 40 00 80 06  0b 12 c0 a8 2b 2a 45 c3   ..?.@... ....+*E.
0020  7c 70 e1 cb 00 50 03 e3  07 22 5f 45 14 e0 50 18   |p...P.. ."_E..P.
0030  11 1c 33 c1 00 00 50 4f  53 54 20 2f 69 6e 64 65   ..3...PO ST /inde
0040  78 2e 70 68 70 20 48 54  54 50 2f 31 2e 31 0d 0a   x.php HT TP/1.1..
```

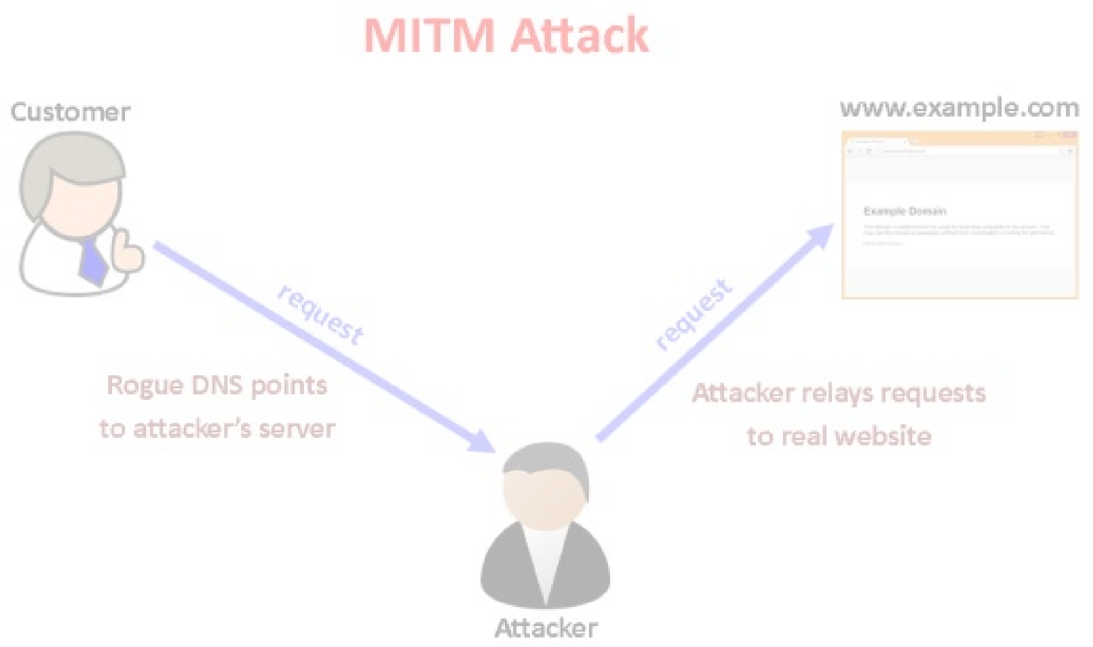
Frame (frame), 724 bytes | Packets: 666 · Disp... | Profile: Default

(bron: guru99)

- Zonder Wifi-beveiliging kan iedereen alle HTTP requests zien

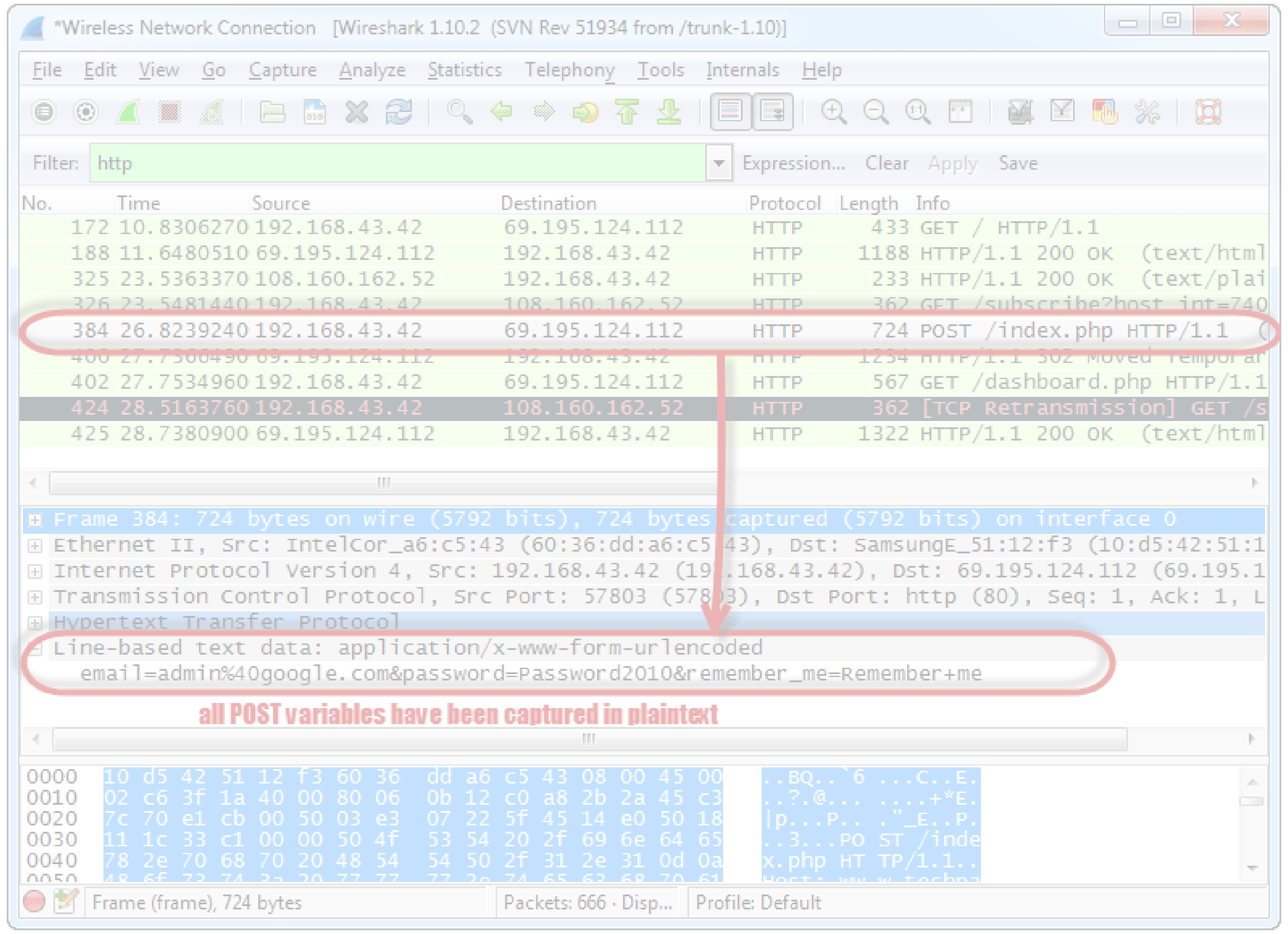
TWEE PROBLEMEN MET HTTP

Man in the middle attack


MITM Attack
Customer
www.example.com
Example Domain
request
request
Rogue DNS points to attacker's server
Attacker relays requests to real website
Attacker

- Certificaat nodig om te bewijzen dat jij de échte server bent.
- Voorbeeld: Russische aanval vanuit een auto buiten het OPCW gebouw in Scheveningen

Stiekem meeluisteren


*Wireless Network Connection [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: http
Expression... Clear Apply Save

```
No.   Time        Source          Destination      Protocol Length Info
  172 10.8306270 192.168.43.42   69.195.124.112   HTTP      433 GET / HTTP/1.1
  188 11.6480510 69.195.124.112  192.168.43.42    HTTP     1188 HTTP/1.1 200 OK  (text/html
  325 23.5363370 108.160.162.52  192.168.43.42    HTTP      233 HTTP/1.1 200 OK  (text/plai
  326 23.5481440 192.168.43.42   108.160.162.52   HTTP      362 GET /subscribe?host_int=740
  384 26.8239240 192.168.43.42   69.195.124.112   HTTP      724 POST /index.php HTTP/1.1
  400 27.7300490 69.195.124.112  192.168.43.42    HTTP     1234 HTTP/1.1 302 Moved Temporar
  402 27.7534960 192.168.43.42   69.195.124.112   HTTP      567 GET /dashboard.php HTTP/1.1
  424 28.5163760 192.168.43.42   108.160.162.52   HTTP      362 [TCP Retransmission] GET /s
  425 28.7380900 69.195.124.112  192.168.43.42    HTTP     1322 HTTP/1.1 200 OK  (text/html

⊞ Frame 384: 724 bytes on wire (5792 bits), 724 bytes captured (5792 bits) on interface 0
⊞ Ethernet II, Src: IntelCor_a6:c5:43 (60:36:dd:a6:c5:43), Dst: SamsungE_51:12:f3 (10:d5:42:51:1
⊞ Internet Protocol Version 4, Src: 192.168.43.42 (192.168.43.42), Dst: 69.195.124.112 (69.195.1
⊞ Transmission Control Protocol, Src Port: 57803 (57803), Dst Port: http (80), Seq: 1, Ack: 1, L
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: application/x-www-form-urlencoded
    email=admin%40google.com&password=Password2010&remember_me=Remember+me

0000  10 d5 42 51 12 f3 60 36  dd a6 c5 43 08 00 45 00   ..BQ..`6 ...C..E.
0010  02 c6 3f 1a 40 00 80 06  0b 12 c0 a8 2b 2a 45 c3   ..?.@... ....+*E.
0020  7c 70 e1 cb 00 50 03 e3  07 22 5f 45 14 e0 50 18   |p...P.. ."_E..P.
0030  11 1c 33 c1 00 00 50 4f  53 54 20 2f 69 6e 64 65   ..3...PO ST /inde
0040  78 2e 70 68 70 20 48 54  54 50 2f 31 2e 31 0d 0a   x.php HT TP/1.1..
```

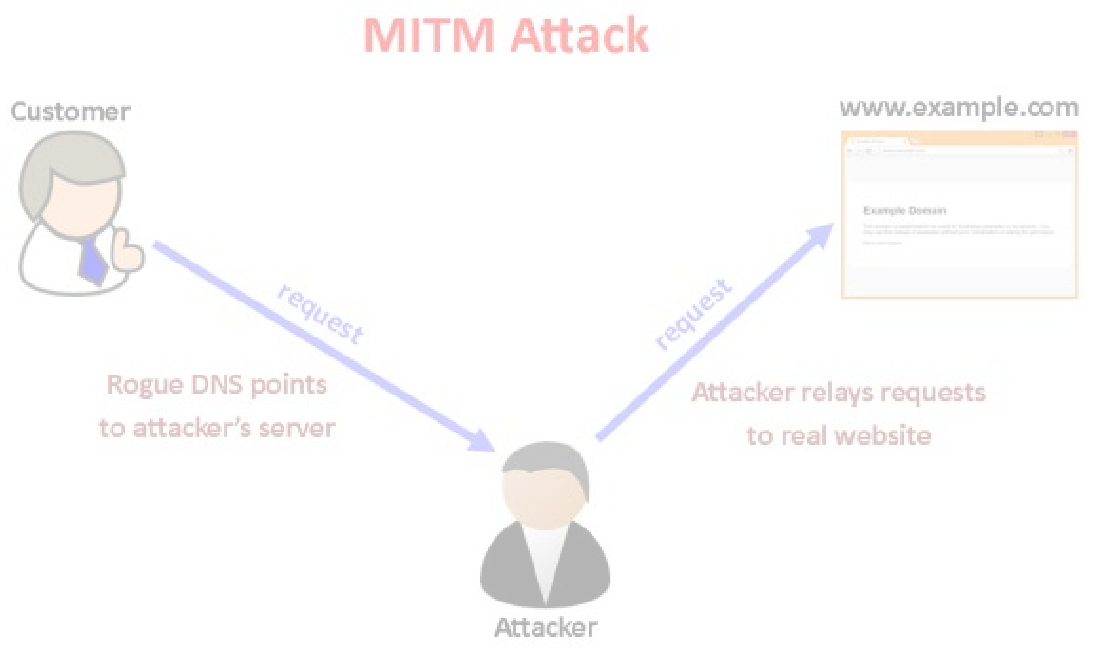
all POST variables have been captured in plaintext

(bron: guru99)

- Zonder Wifi-beveiliging kan iedereen alle HTTP requests zien

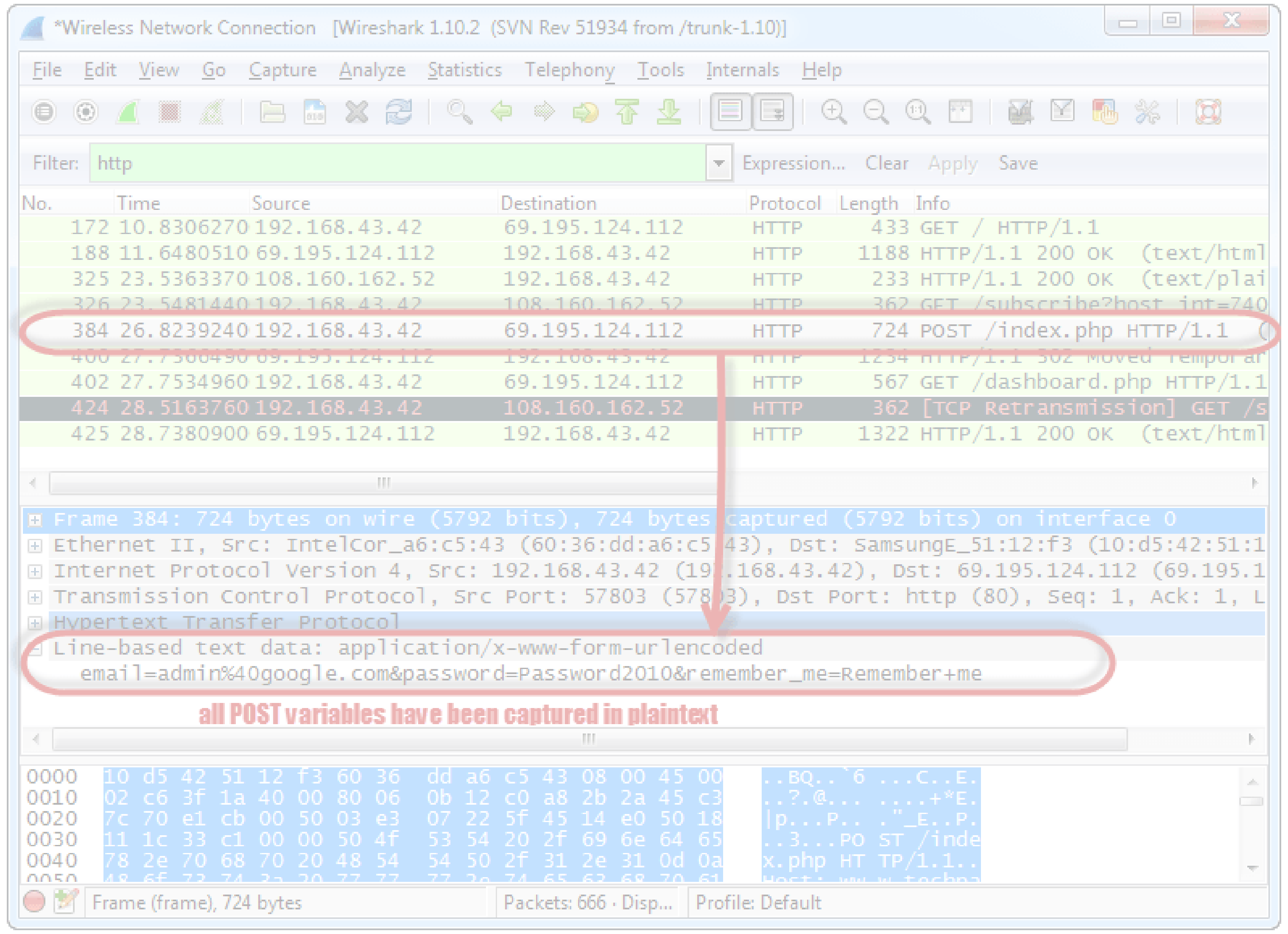
TWEE PROBLEMEN MET HTTP

Man in the middle attack



- Certificaat nodig om te bewijzen dat jij de échte server bent.
- Voorbeeld: Russische aanval vanuit een auto buiten het OPCW gebouw in Scheveningen

Stiekem meeluisteren


*Wireless Network Connection [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: http
Expression... Clear Apply Save
all POST variables have been captured in plaintext

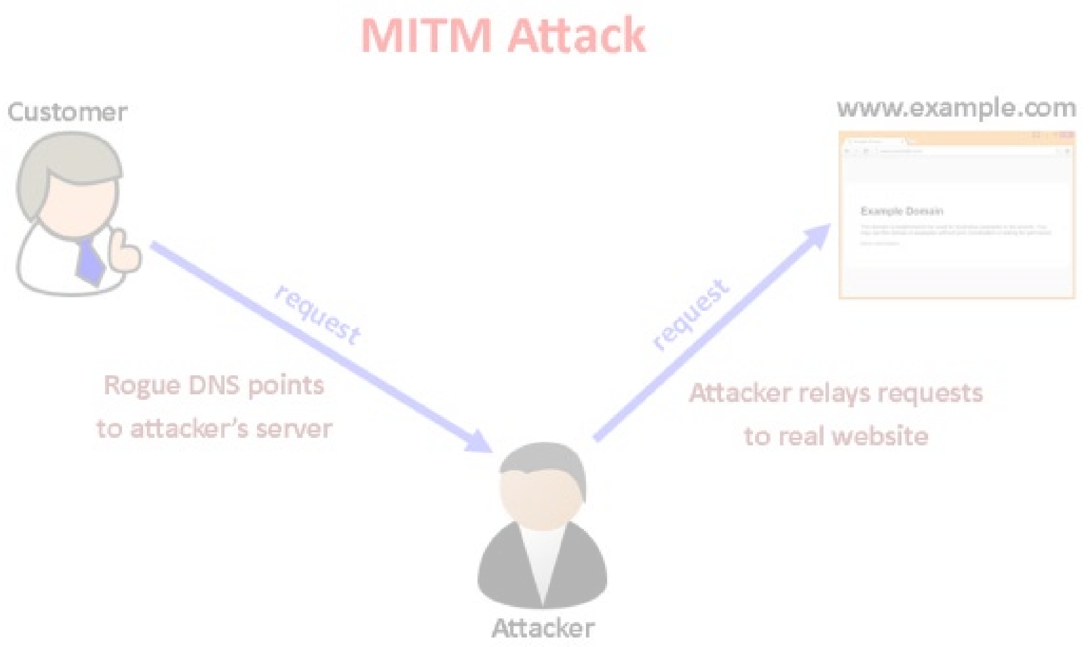
```
No.   Time       Source         Destination     Protocol Length Info
  172 10.8306270 192.168.43.42  69.195.124.112  HTTP      433 GET / HTTP/1.1
  188 11.6480510 69.195.124.112 192.168.43.42   HTTP     1188 HTTP/1.1 200 OK  (text/html
  325 23.5363370 108.160.162.52 192.168.43.42   HTTP      233 HTTP/1.1 200 OK  (text/plai
  326 23.5481440 192.168.43.42  108.160.162.52  HTTP      362 GET /subscribe?host_int=740
  384 26.8239240 192.168.43.42  69.195.124.112  HTTP      724 POST /index.php HTTP/1.1
  400 27.7300490 69.195.124.112 192.168.43.42   HTTP     1234 HTTP/1.1 302 Moved Temporar
  402 27.7534960 192.168.43.42  69.195.124.112  HTTP      567 GET /dashboard.php HTTP/1.1
  424 28.5163760 192.168.43.42  108.160.162.52  HTTP      362 [TCP Retransmission] GET /s
  425 28.7380900 69.195.124.112 192.168.43.42   HTTP     1322 HTTP/1.1 200 OK  (text/html

⊞ Frame 384: 724 bytes on wire (5792 bits), 724 bytes captured (5792 bits) on interface 0
⊞ Ethernet II, Src: IntelCor_a6:c5:43 (60:36:dd:a6:c5:43), Dst: SamsungE_51:12:f3 (10:d5:42:51:1
⊞ Internet Protocol Version 4, Src: 192.168.43.42 (192.168.43.42), Dst: 69.195.124.112 (69.195.1
⊞ Transmission Control Protocol, Src Port: 57803 (57803), Dst Port: http (80), Seq: 1, Ack: 1, L
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: application/x-www-form-urlencoded
    email=admin%40google.com&password=Password2010&remember_me=Remember+me

0000  10 d5 42 51 12 f3 60 36  dd a6 c5 43 08 00 45 00   ..BQ..`6 ...C..E.
0010  02 c6 3f 1a 40 00 80 06  0b 12 c0 a8 2b 2a 45 c3   ..?.@... ....+*E.
0020  7c 70 e1 cb 00 50 03 e3  07 22 5f 45 14 e0 50 18   |p...P.. ."_E..P.
0030  11 1c 33 c1 00 00 50 4f  53 54 20 2f 69 6e 64 65   ..3...PO ST /inde
0040  78 2e 70 68 70 20 48 54  54 50 2f 31 2e 31 0d 0a   x.php HT TP/1.1..

Frame (frame), 724 bytes    Packets: 666 · Disp...    Profile: Default
```

(bron: guru99)

- Zonder Wifi-beveiliging kan iedereen alle HTTP requests zien

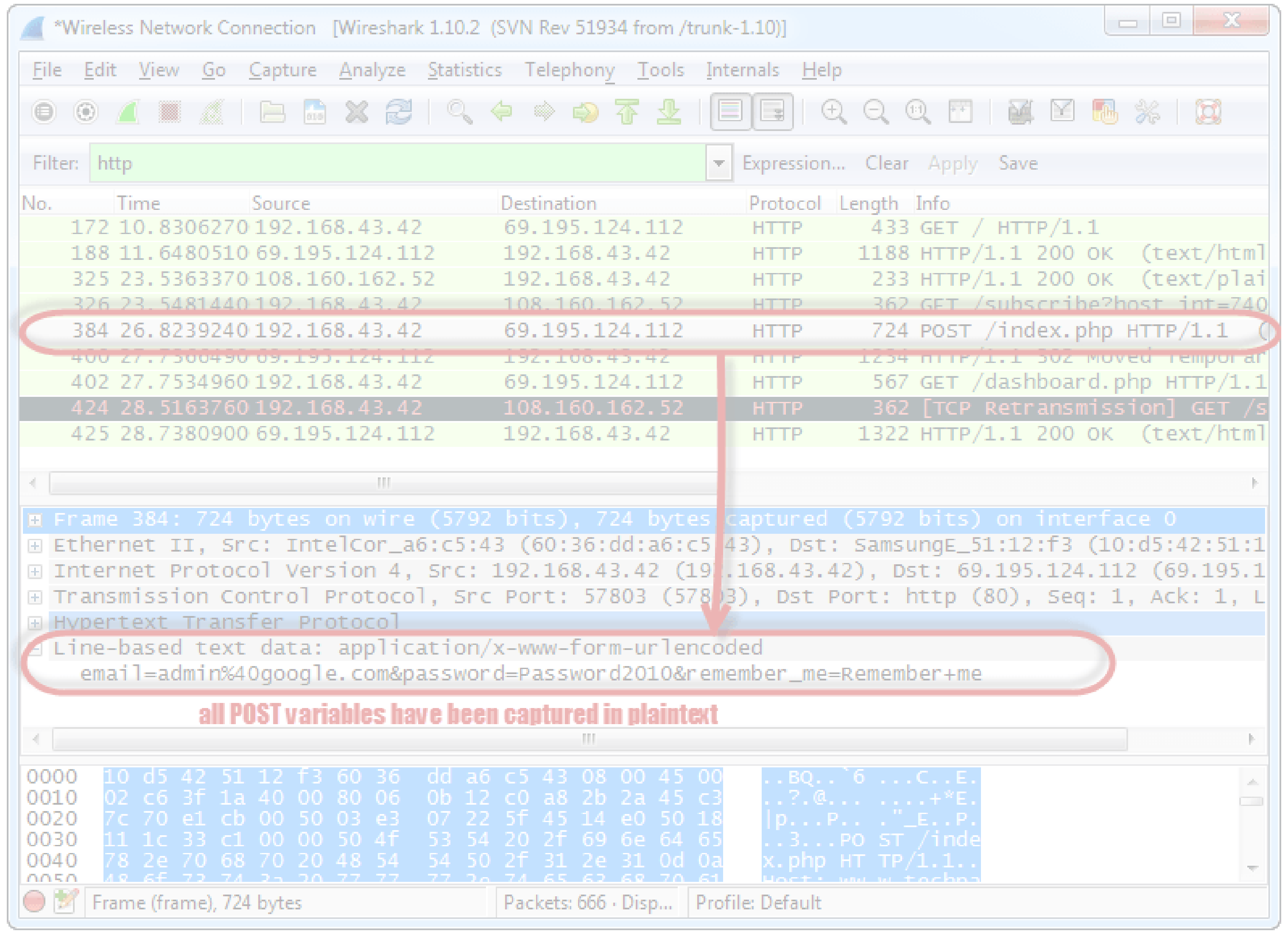
TWEE PROBLEMEN MET HTTP

Man in the middle attack



- Certificaat nodig om te bewijzen dat jij de échte server bent.
- Voorbeeld: Russische aanval vanuit een auto buiten het OPCW gebouw in Scheveningen

Stiekem meeluisteren


*Wireless Network Connection [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save

Filter: http

```
No.     Time       Source          Destination      Protocol  Length Info
    172 10.8306270 192.168.43.42   69.195.124.112   HTTP       433 GET / HTTP/1.1
    188 11.6480510 69.195.124.112  192.168.43.42    HTTP      1188 HTTP/1.1 200 OK  (text/html
    325 23.5363370 108.160.162.52  192.168.43.42    HTTP       233 HTTP/1.1 200 OK  (text/plai
    326 23.5481440 192.168.43.42   108.160.162.52   HTTP       362 GET /subscribe?host_int=740
    384 26.8239240 192.168.43.42   69.195.124.112   HTTP       724 POST /index.php HTTP/1.1
    400 27.7300490 69.195.124.112  192.168.43.42    HTTP      1234 HTTP/1.1 302 Moved Temporar
    402 27.7534960 192.168.43.42   69.195.124.112   HTTP       567 GET /dashboard.php HTTP/1.1
    424 28.5163760 192.168.43.42   108.160.162.52   HTTP       362 [TCP Retransmission] GET /s
    425 28.7380900 69.195.124.112  192.168.43.42    HTTP      1322 HTTP/1.1 200 OK  (text/html

⊞ Frame 384: 724 bytes on wire (5792 bits), 724 bytes captured (5792 bits) on interface 0
⊞ Ethernet II, Src: IntelCor_a6:c5:43 (60:36:dd:a6:c5:43), Dst: SamsungE_51:12:f3 (10:d5:42:51:1
⊞ Internet Protocol Version 4, Src: 192.168.43.42 (192.168.43.42), Dst: 69.195.124.112 (69.195.1
⊞ Transmission Control Protocol, Src Port: 57803 (57803), Dst Port: http (80), Seq: 1, Ack: 1, L
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: application/x-www-form-urlencoded
    email=admin%40google.com&password=Password2010&remember_me=Remember+me

0000  10 d5 42 51 12 f3 60 36  dd a6 c5 43 08 00 45 00   ..BQ..`6 ...C..E.
0010  02 c6 3f 1a 40 00 80 06  0b 12 c0 a8 2b 2a 45 c3   ..?.@... ....+*E.
0020  7c 70 e1 cb 00 50 03 e3  07 22 5f 45 14 e0 50 18   |p...P.. ."_E..P.
0030  11 1c 33 c1 00 00 50 4f  53 54 20 2f 69 6e 64 65   ..3...PO ST /inde
0040  78 2e 70 68 70 20 48 54  54 50 2f 31 2e 31 0d 0a   x.php HT TP/1.1..
```

all POST variables have been captured in plaintext

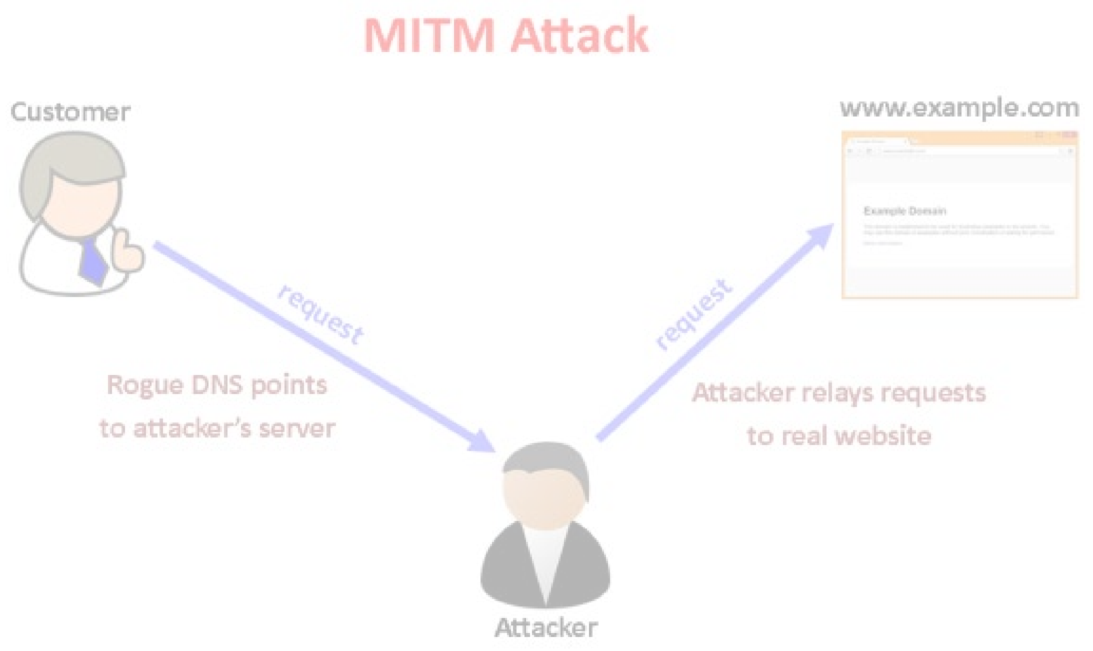
Frame (frame), 724 bytes | Packets: 666 · Disp... | Profile: Default

(bron: guru99)

- Zonder Wifi-beveiliging kan iedereen alle HTTP requests zien

TWEE PROBLEMEN MET HTTP

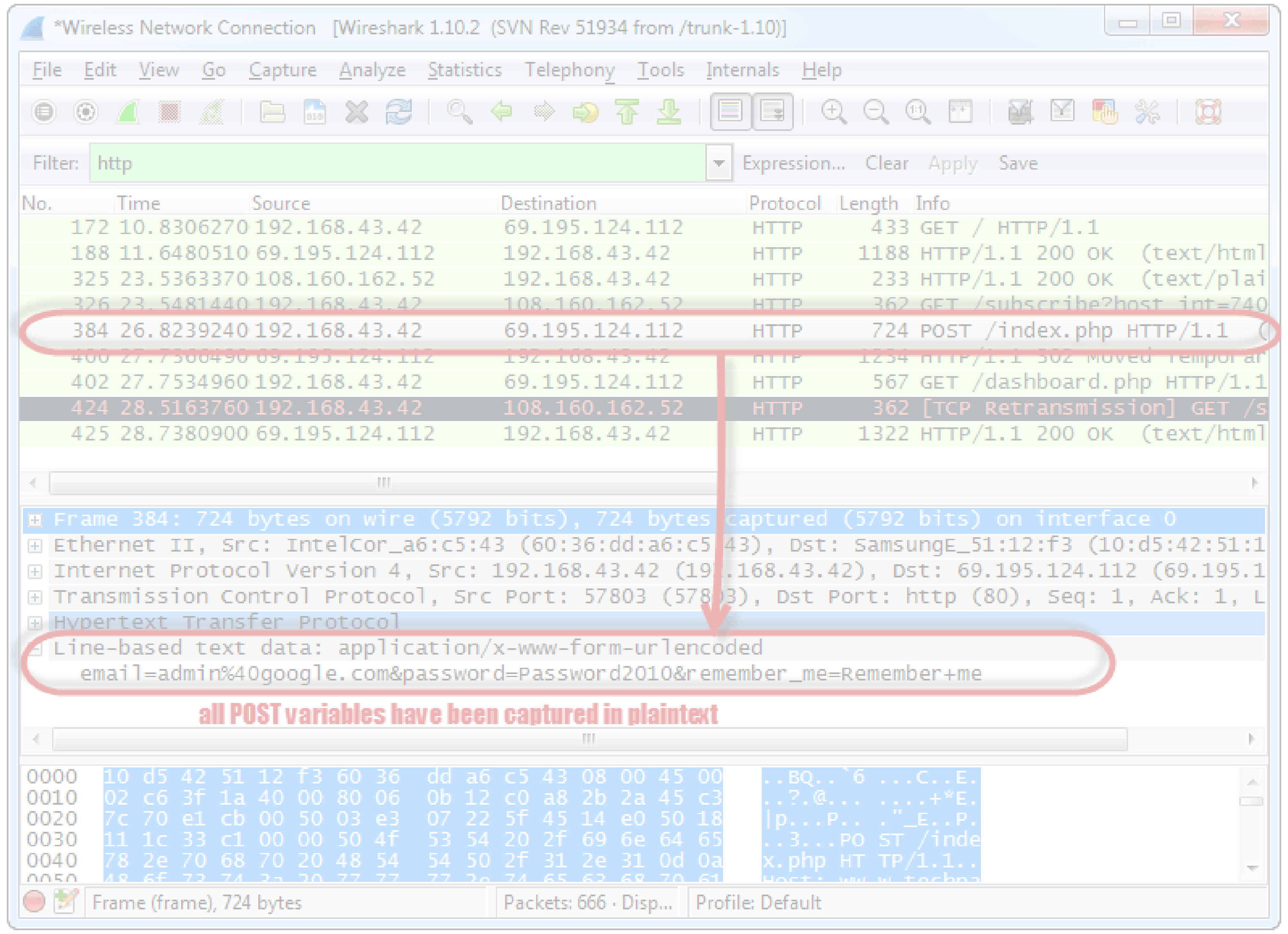
Man in the middle attack



MITM Attack
Customer
www.example.com
Example Domain
request
request
Rogue DNS points to attacker's server
Attacker relays requests to real website
Attacker

- Certificaat nodig om te bewijzen dat jij de échte server bent.
- Voorbeeld: Russische aanval vanuit een auto buiten het OPCW gebouw in Scheveningen

Stiekem meeluisteren



*Wireless Network Connection [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: http
Expression... Clear Apply Save

```
No.   Time        Source          Destination     Protocol Length Info
  172 10.8306270  192.168.43.42   69.195.124.112  HTTP      433 GET / HTTP/1.1
  188 11.6480510  69.195.124.112  192.168.43.42   HTTP     1188 HTTP/1.1 200 OK  (text/html
  325 23.5363370  108.160.162.52  192.168.43.42   HTTP      233 HTTP/1.1 200 OK  (text/plai
  326 23.5481440  192.168.43.42   108.160.162.52  HTTP      362 GET /subscribe?host_int=740
  384 26.8239240  192.168.43.42   69.195.124.112  HTTP      724 POST /index.php HTTP/1.1  (
  400 27.7300490  69.195.124.112  192.168.43.42   HTTP     1234 HTTP/1.1 302 Moved Temporar
  402 27.7534960  192.168.43.42   69.195.124.112  HTTP      567 GET /dashboard.php HTTP/1.1
  424 28.5163760  192.168.43.42   108.160.162.52  HTTP      362 [TCP Retransmission] GET /s
  425 28.7380900  69.195.124.112  192.168.43.42   HTTP     1322 HTTP/1.1 200 OK  (text/html
```

```
⊞ Frame 384: 724 bytes on wire (5792 bits), 724 bytes captured (5792 bits) on interface 0
⊞ Ethernet II, Src: IntelCor_a6:c5:43 (60:36:dd:a6:c5:43), Dst: SamsungE_51:12:f3 (10:d5:42:51:1
⊞ Internet Protocol Version 4, Src: 192.168.43.42 (192.168.43.42), Dst: 69.195.124.112 (69.195.1
⊞ Transmission Control Protocol, Src Port: 57803 (57803), Dst Port: http (80), Seq: 1, Ack: 1, L
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: application/x-www-form-urlencoded
    email=admin%40google.com&password=Password2010&remember_me=Remember+me
```

all POST variables have been captured in plaintext

```
0000  10 d5 42 51 12 f3 60 36  dd a6 c5 43 08 00 45 00   ..BQ..`6 ...C..E.
0010  02 c6 3f 1a 40 00 80 06  0b 12 c0 a8 2b 2a 45 c3   ..?.@... ....+*E.
0020  7c 70 e1 cb 00 50 03 e3  07 22 5f 45 14 e0 50 18   |p...P.. ."_E..P.
0030  11 1c 33 c1 00 00 50 4f  53 54 20 2f 69 6e 64 65   ..3...PO ST /inde
0040  78 2e 70 68 70 20 48 54  54 50 2f 31 2e 31 0d 0a   x.php HT TP/1.1..
0050  48 6f 73 74 3a 20 77 77  77 2e 74 65 63 68 70 61   Host: ww w.techpa
```

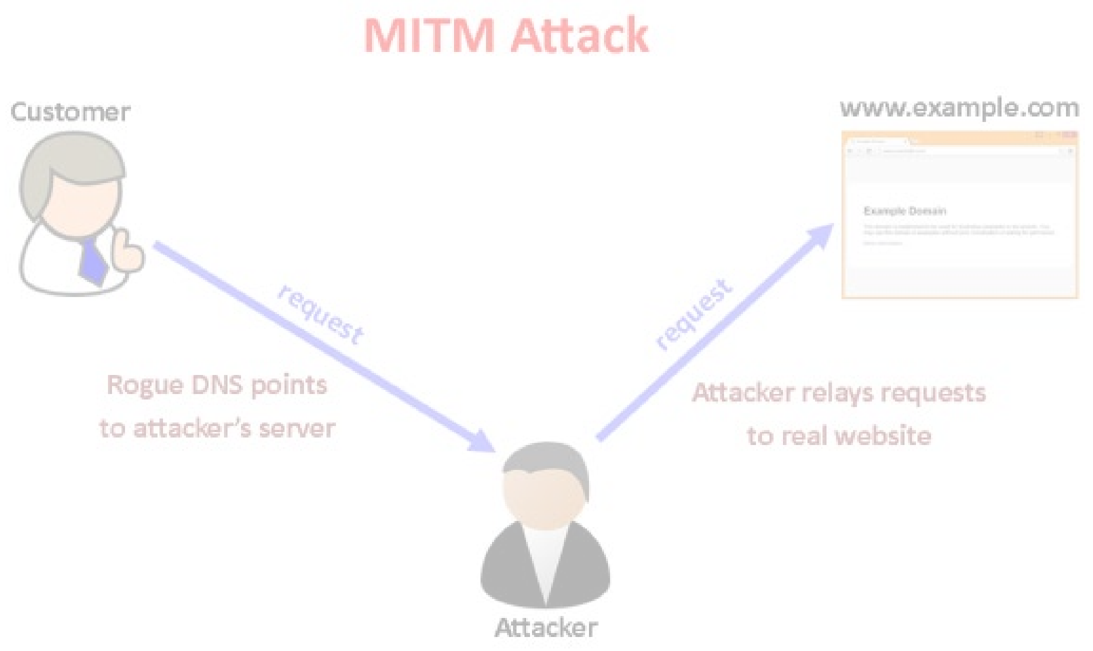
Frame (frame), 724 bytes | Packets: 666 · Disp... | Profile: Default

(bron: guru99)

- Zonder Wifi-beveiliging kan iedereen alle HTTP requests zien

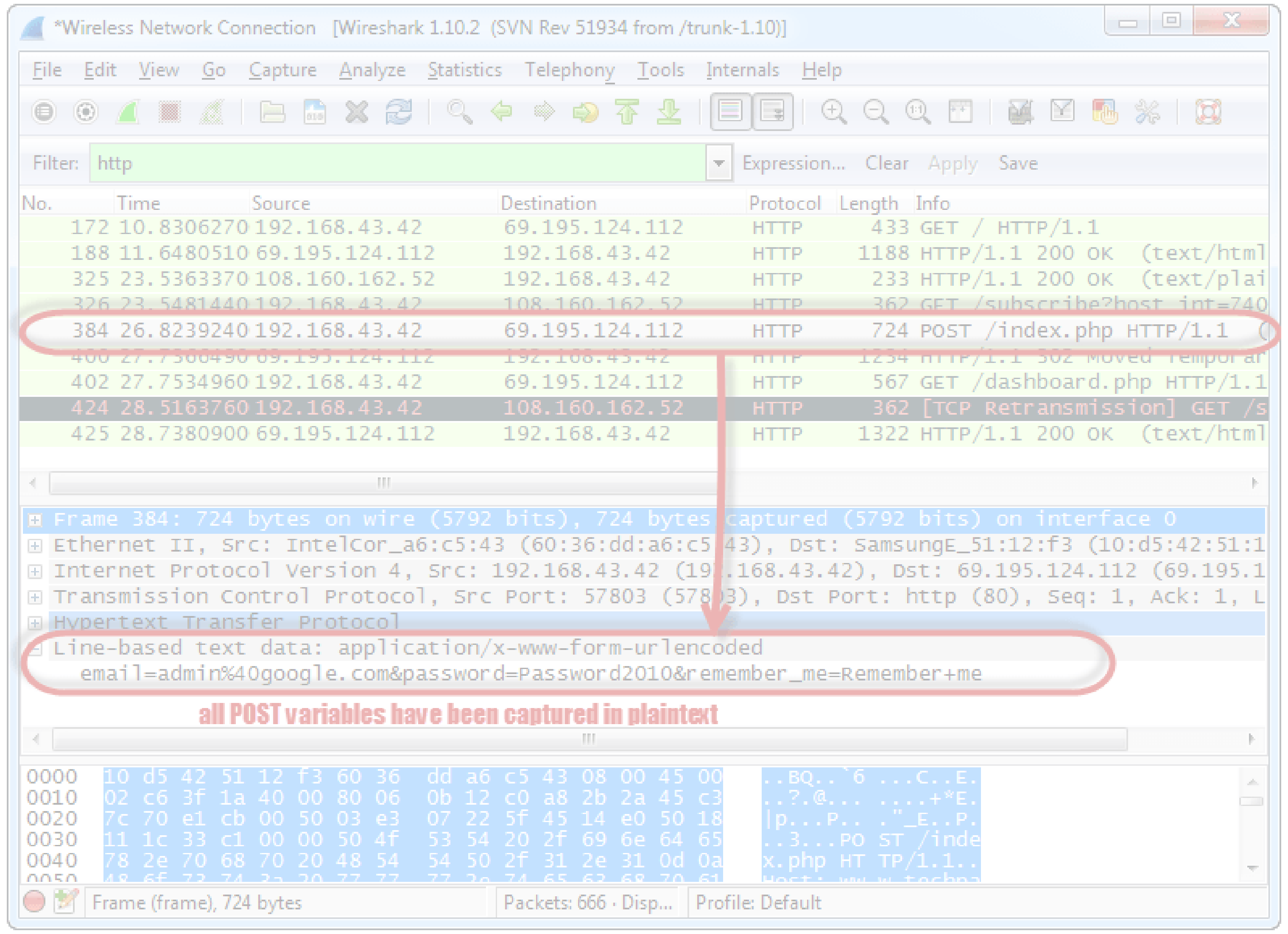
TWEE PROBLEMEN MET HTTP

Man in the middle attack



- Certificaat nodig om te bewijzen dat jij de échte server bent.
- Voorbeeld: Russische aanval vanuit een auto buiten het OPCW gebouw in Scheveningen

Stiekem meeluisteren


*Wireless Network Connection [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: http
Expression... Clear Apply Save

```
No.   Time       Source          Destination     Protocol Length Info
  172 10.8306270 192.168.43.42   69.195.124.112  HTTP      433 GET / HTTP/1.1
  188 11.6480510 69.195.124.112  192.168.43.42   HTTP     1188 HTTP/1.1 200 OK  (text/html
  325 23.5363370 108.160.162.52  192.168.43.42   HTTP      233 HTTP/1.1 200 OK  (text/plai
  326 23.5481440 192.168.43.42   108.160.162.52  HTTP      362 GET /subscribe?host_int=740
  384 26.8239240 192.168.43.42   69.195.124.112  HTTP      724 POST /index.php HTTP/1.1  (
  400 27.7300490 69.195.124.112  192.168.43.42   HTTP     1234 HTTP/1.1 302 Moved Temporar
  402 27.7534960 192.168.43.42   69.195.124.112  HTTP      567 GET /dashboard.php HTTP/1.1
  424 28.5163760 192.168.43.42   108.160.162.52  HTTP      362 [TCP Retransmission] GET /s
  425 28.7380900 69.195.124.112  192.168.43.42   HTTP     1322 HTTP/1.1 200 OK  (text/html

⊞ Frame 384: 724 bytes on wire (5792 bits), 724 bytes captured (5792 bits) on interface 0
⊞ Ethernet II, Src: IntelCor_a6:c5:43 (60:36:dd:a6:c5:43), Dst: SamsungE_51:12:f3 (10:d5:42:51:1
⊞ Internet Protocol Version 4, Src: 192.168.43.42 (192.168.43.42), Dst: 69.195.124.112 (69.195.1
⊞ Transmission Control Protocol, Src Port: 57803 (57803), Dst Port: http (80), Seq: 1, Ack: 1, L
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: application/x-www-form-urlencoded
    email=admin%40google.com&password=Password2010&remember_me=Remember+me

0000  10 d5 42 51 12 f3 60 36  dd a6 c5 43 08 00 45 00   ..BQ..`6 ...C..E.
0010  02 c6 3f 1a 40 00 80 06  0b 12 c0 a8 2b 2a 45 c3   ..?.@... ....+*E.
0020  7c 70 e1 cb 00 50 03 e3  07 22 5f 45 14 e0 50 18   |p...P.. ."_E..P.
0030  11 1c 33 c1 00 00 50 4f  53 54 20 2f 69 6e 64 65   ..3...PO ST /inde
0040  78 2e 70 68 70 20 48 54  54 50 2f 31 2e 31 0d 0a   x.php HT TP/1.1..

Frame (frame), 724 bytes | Packets: 666 · Disp... | Profile: Default
```

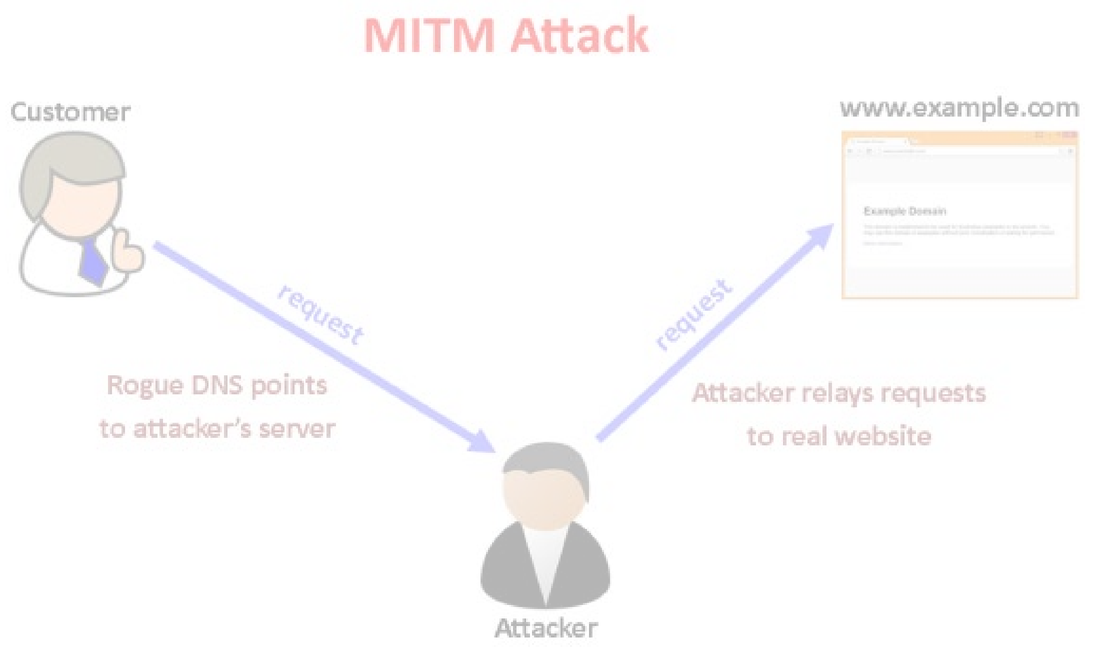
all POST variables have been captured in plaintext

(bron: guru99)

- Zonder Wifi-beveiliging kan iedereen alle HTTP requests zien

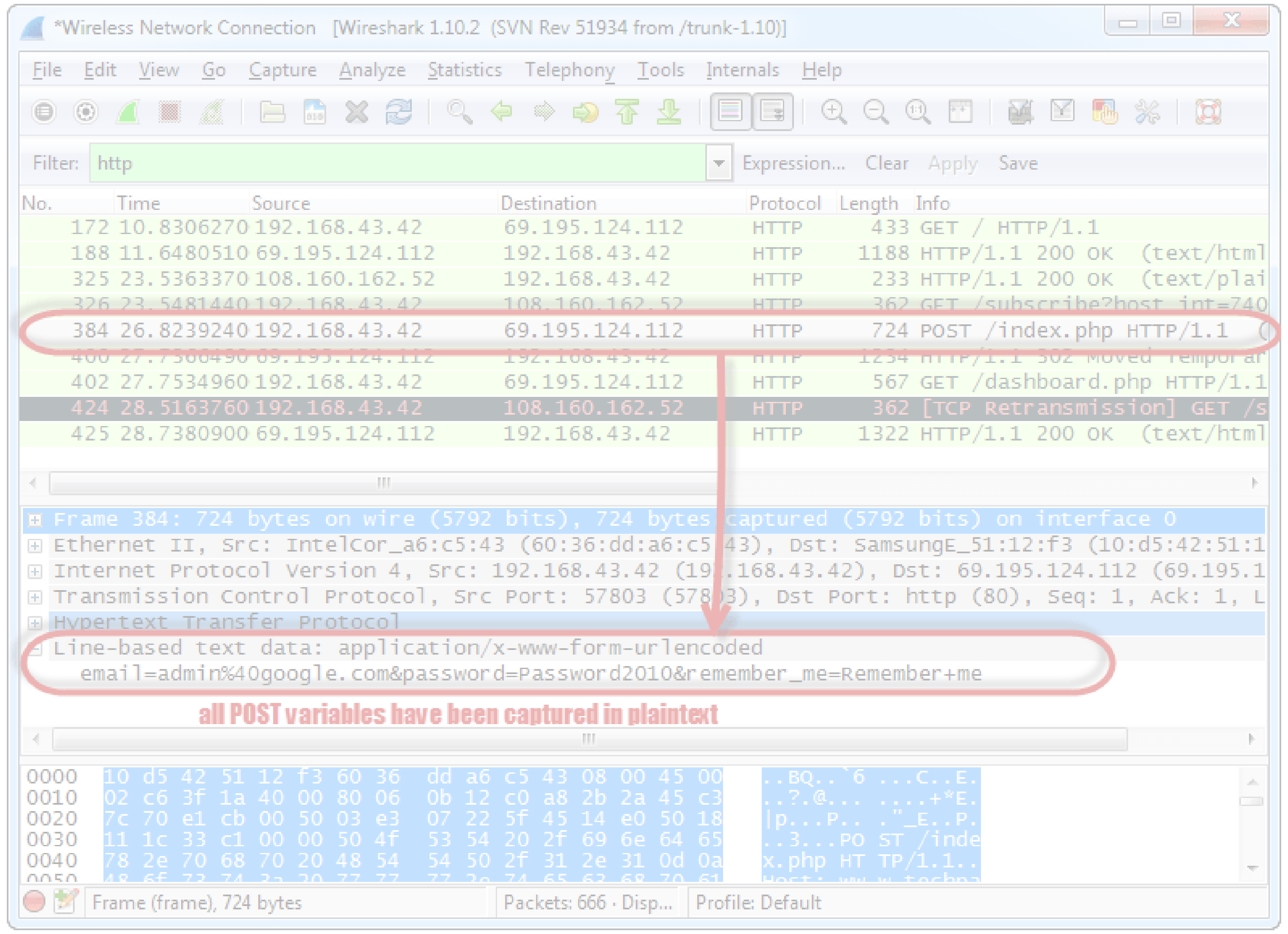
TWEE PROBLEMEN MET HTTP

Man in the middle attack



- Certificaat nodig om te bewijzen dat jij de échte server bent.
- Voorbeeld: Russische aanval vanuit een auto buiten het OPCW gebouw in Scheveningen

Stiekem meeluisteren


*Wireless Network Connection [Wireshark 1.10.2 (SVN Rev 51934 from /trunk-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save

Filter: http

```
No.     Time       Source          Destination     Protocol Length Info
    172 10.8306270 192.168.43.42   69.195.124.112  HTTP      433 GET / HTTP/1.1
    188 11.6480510 69.195.124.112  192.168.43.42   HTTP     1188 HTTP/1.1 200 OK  (text/html
    325 23.5363370 108.160.162.52  192.168.43.42   HTTP      233 HTTP/1.1 200 OK  (text/plai
    326 23.5481440 192.168.43.42   108.160.162.52  HTTP      362 GET /subscribe?host_int=740
    384 26.8239240 192.168.43.42   69.195.124.112  HTTP      724 POST /index.php HTTP/1.1  (
    400 27.7300490 69.195.124.112  192.168.43.42   HTTP     1234 HTTP/1.1 302 Moved Temporar
    402 27.7534960 192.168.43.42   69.195.124.112  HTTP      567 GET /dashboard.php HTTP/1.1
    424 28.5163760 192.168.43.42   108.160.162.52  HTTP      362 [TCP Retransmission] GET /s
    425 28.7380900 69.195.124.112  192.168.43.42   HTTP     1322 HTTP/1.1 200 OK  (text/html

⊞ Frame 384: 724 bytes on wire (5792 bits), 724 bytes captured (5792 bits) on interface 0
⊞ Ethernet II, Src: IntelCor_a6:c5:43 (60:36:dd:a6:c5:43), Dst: SamsungE_51:12:f3 (10:d5:42:51:1
⊞ Internet Protocol Version 4, Src: 192.168.43.42 (192.168.43.42), Dst: 69.195.124.112 (69.195.1
⊞ Transmission Control Protocol, Src Port: 57803 (57803), Dst Port: http (80), Seq: 1, Ack: 1, L
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: application/x-www-form-urlencoded
    email=admin%40google.com&password=Password2010&remember_me=Remember+me

0000  10 d5 42 51 12 f3 60 36  dd a6 c5 43 08 00 45 00   ..BQ..`6 ...C..E.
0010  02 c6 3f 1a 40 00 80 06  0b 12 c0 a8 2b 2a 45 c3   ..?.@... ....+*E.
0020  7c 70 e1 cb 00 50 03 e3  07 22 5f 45 14 e0 50 18   |p...P.. ."_E..P.
0030  11 1c 33 c1 00 00 50 4f  53 54 20 2f 69 6e 64 65   ..3...PO ST /inde
0040  78 2e 70 68 70 20 48 54  54 50 2f 31 2e 31 0d 0a   x.php HT TP/1.1..

Frame (frame), 724 bytes    Packets: 666 · Disp...    Profile: Default
```

all POST variables have been captured in plaintext

(bron: guru99)

- Zonder Wifi-beveiliging kan iedereen alle HTTP requests zien

🛡 HTTPS

- Hypertext Transfer Protocol **Secure**
- Een heleboel is HTTPS
- In de HTTPS handshake wordt (ongeveer)
  - de encryptie ingesteld: sleutels gedeeld, ...
    - Alles is daarna geencrypt: de URL, de HTTP headers en de POST data
  - een certificaat van de server naar de client gestuurd
    - self-signed of van een CA
    - de client weet nu zeker dat het met de server praat die eigenaar is van het domein

🛡 HTTPS

- Hypertext Transfer Protocol **Secure**
- Een heleboel is HTTPS
- In de HTTPS handshake wordt (ongeveer)
  - de encryptie ingesteld: sleutels gedeeld, ...
    - Alles is daarna geencrypt: de URL, de HTTP headers en de POST data
  - een certificaat van de server naar de client gestuurd
    - self-signed of van een CA
    - de client weet nu zeker dat het met de server praat die eigenaar is van het domein

🛡 HTTPS

- Hypertext Transfer Protocol **Secure**
- Een heleboel is HTTPS
- In de HTTPS handshake wordt (ongeveer)
  - de encryptie ingesteld: sleutels gedeeld, ...
    - Alles is daarna geencrypt: de URL, de HTTP headers en de POST data
  - een certificaat van de server naar de client gestuurd
    - self-signed of van een CA
    - de client weet nu zeker dat het met de server praat die eigenaar is van het domein

🛡 HTTPS

- Hypertext Transfer Protocol **Secure**
- Een heleboel is HTTPS
- In de HTTPS handshake wordt (ongeveer)
  - de encryptie ingesteld: sleutels gedeeld, ...
    - Alles is daarna geencrypt: de URL, de HTTP headers en de POST data
  - een certificaat van de server naar de client gestuurd
    - self-signed of van een CA
    - de client weet nu zeker dat het met de server praat die eigenaar is van het domein

🛡 HTTPS

- Hypertext Transfer Protocol **Secure**
- Een heleboel is HTTPS
- In de HTTPS handshake wordt (ongeveer)
  - de encryptie ingesteld: sleutels gedeeld, ...
    - Alles is daarna geencrypt: de URL, de HTTP headers en de POST data
  - een certificaat van de server naar de client gestuurd
    - self-signed of van een CA
    - de client weet nu zeker dat het met de server praat die eigenaar is van het domein

🛡 HTTPS

- Hypertext Transfer Protocol **Secure**
- Een heleboel is HTTPS
- In de HTTPS handshake wordt (ongeveer)
  - de encryptie ingesteld: sleutels gedeeld, ...
    - Alles is daarna geencrypt: de URL, de HTTP headers en de POST data
  - een certificaat van de server naar de client gestuurd
    - self-signed of van een CA
    - de client weet nu zeker dat het met de server praat die eigenaar is van het domein

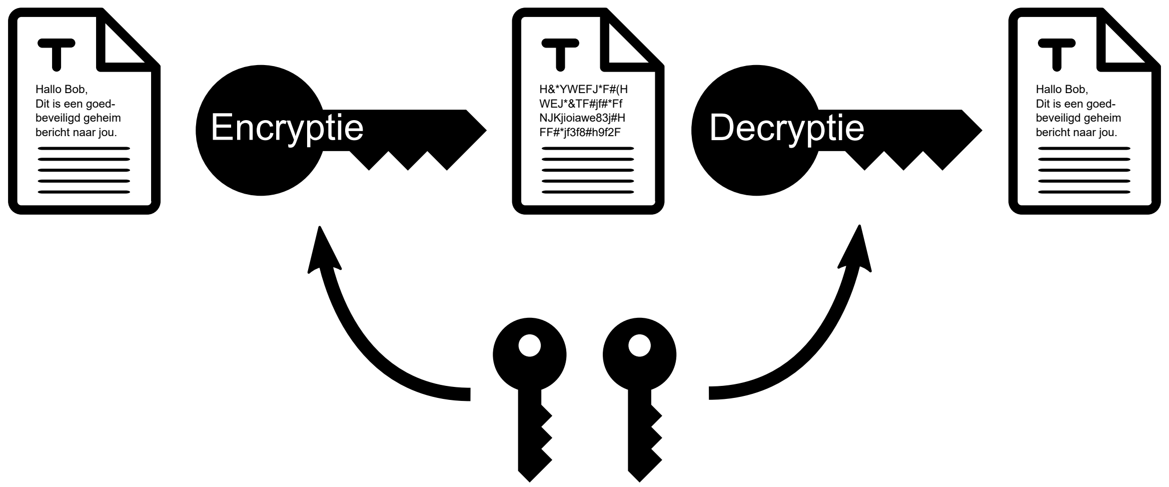
🛡 HTTPS

- Hypertext Transfer Protocol **Secure**
- Een heleboel is HTTPS
- In de HTTPS handshake wordt (ongeveer)
  - de encryptie ingesteld: sleutels gedeeld, ...
    - Alles is daarna geencrypt: de URL, de HTTP headers en de POST data
  - een certificaat van de server naar de client gestuurd
    - self-signed of van een CA
    - de client weet nu zeker dat het met de server praat die eigenaar is van het domein

🛡 HTTPS

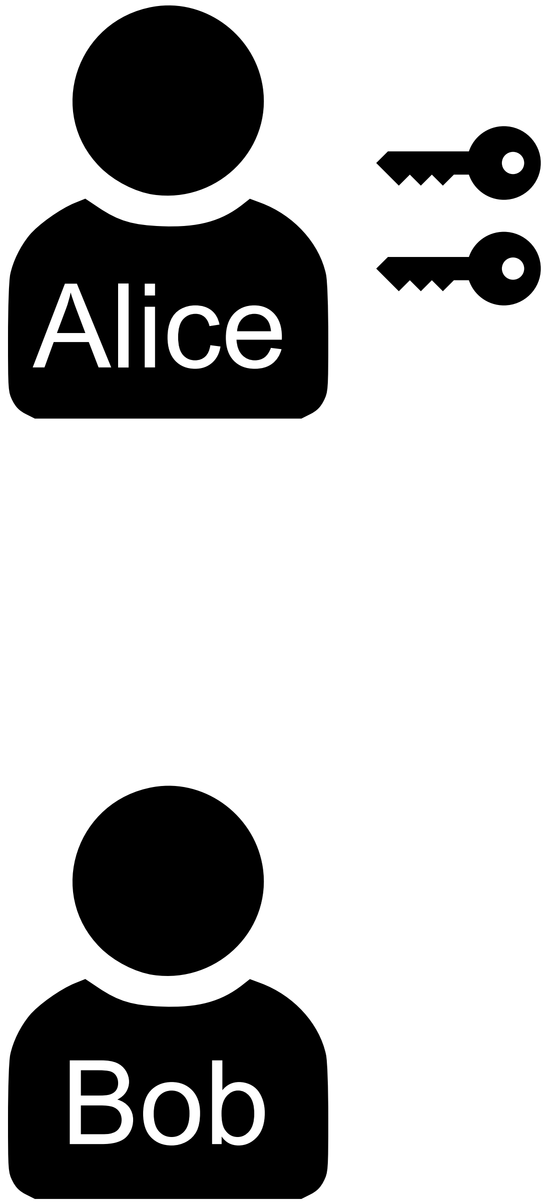
- Hypertext Transfer Protocol **Secure**
- Een heleboel is HTTPS
- In de HTTPS handshake wordt (ongeveer)
  - de encryptie ingesteld: sleutels gedeeld, ...
    - Alles is daarna geencrypt: de URL, de HTTP headers en de POST data
  - een certificaat van de server naar de client gestuurd
    - self-signed of van een CA
    - de client weet nu zeker dat het met de server praat die eigenaar is van het domein

SYMMETRIC KEYS

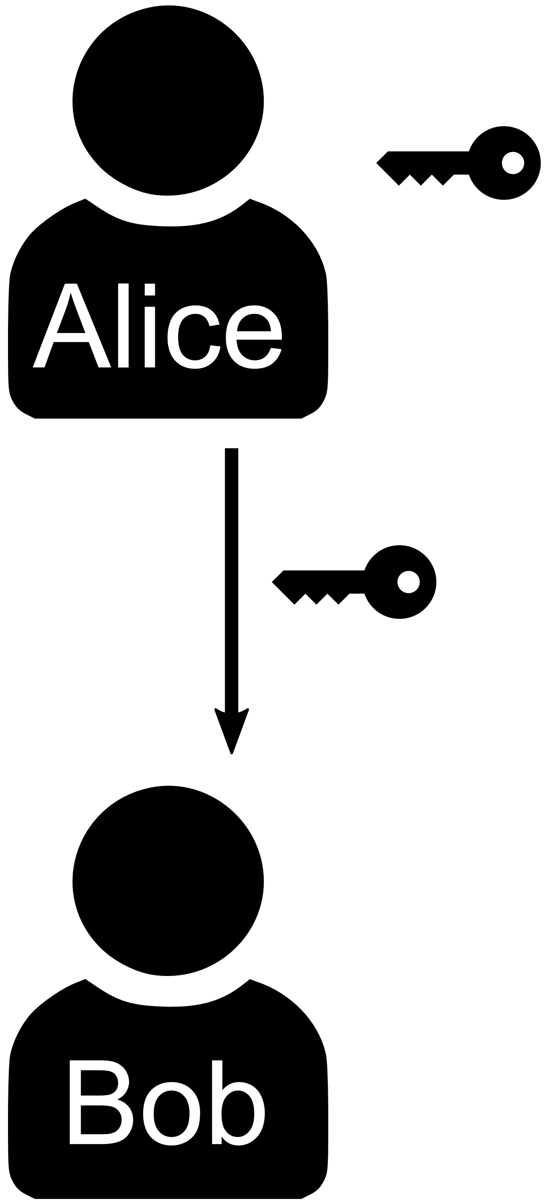


SYMMETRIC KEYS

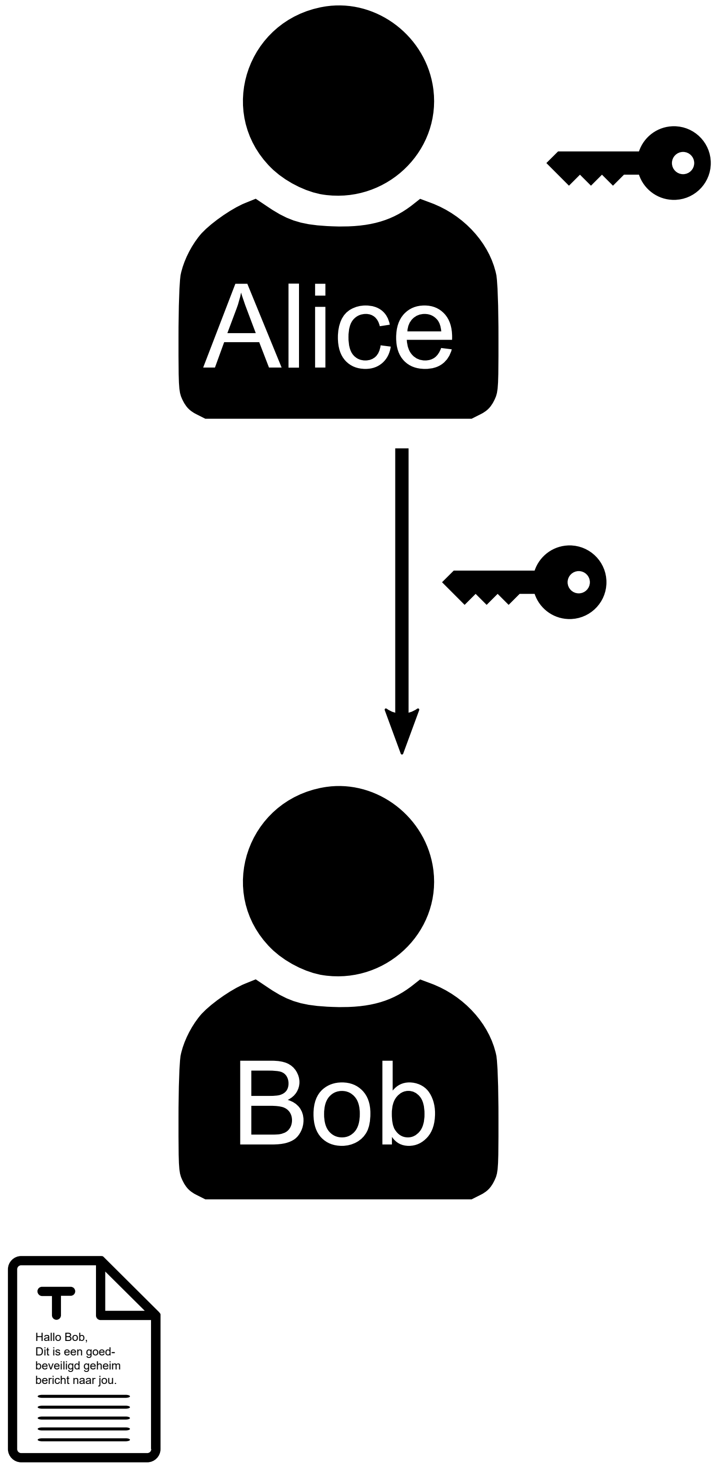
SYMMETRIC KEYS



SYMMETRIC KEYS

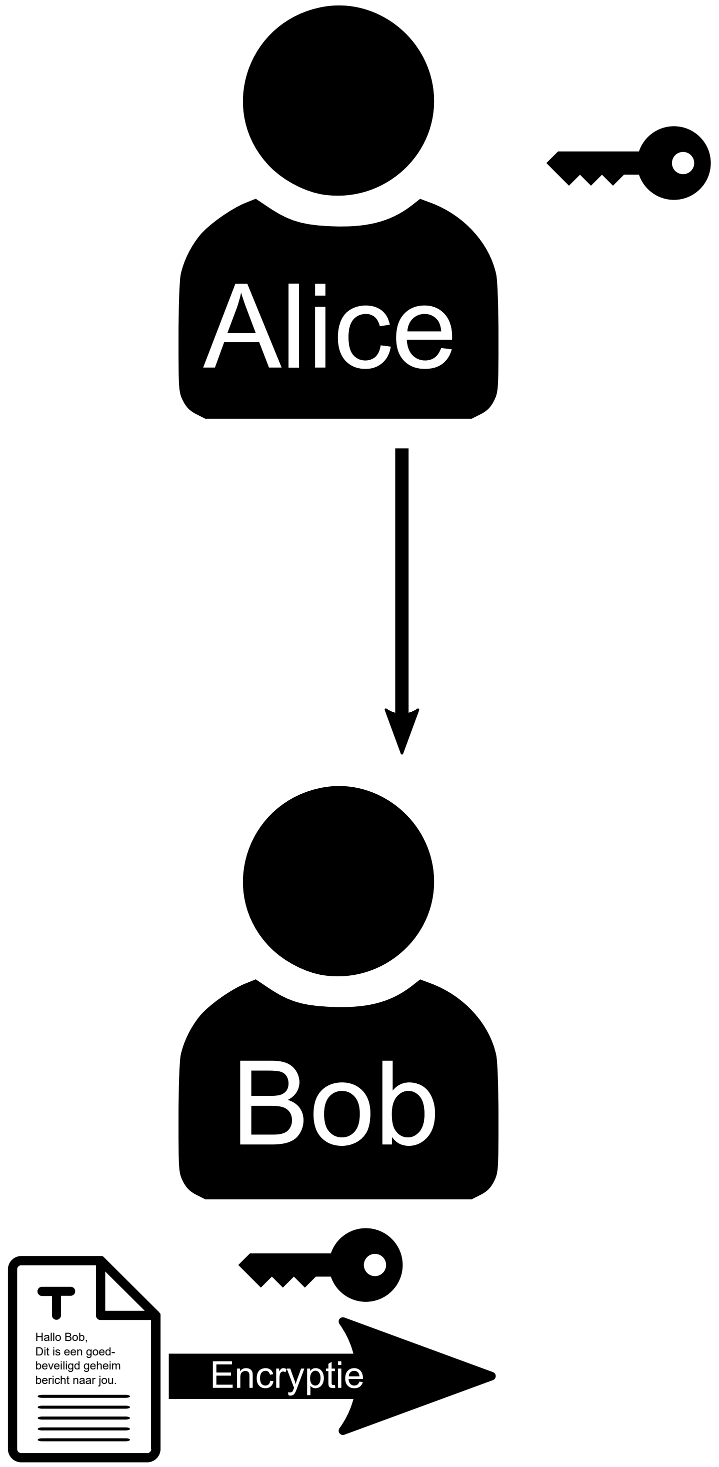


SYMMETRIC KEYS



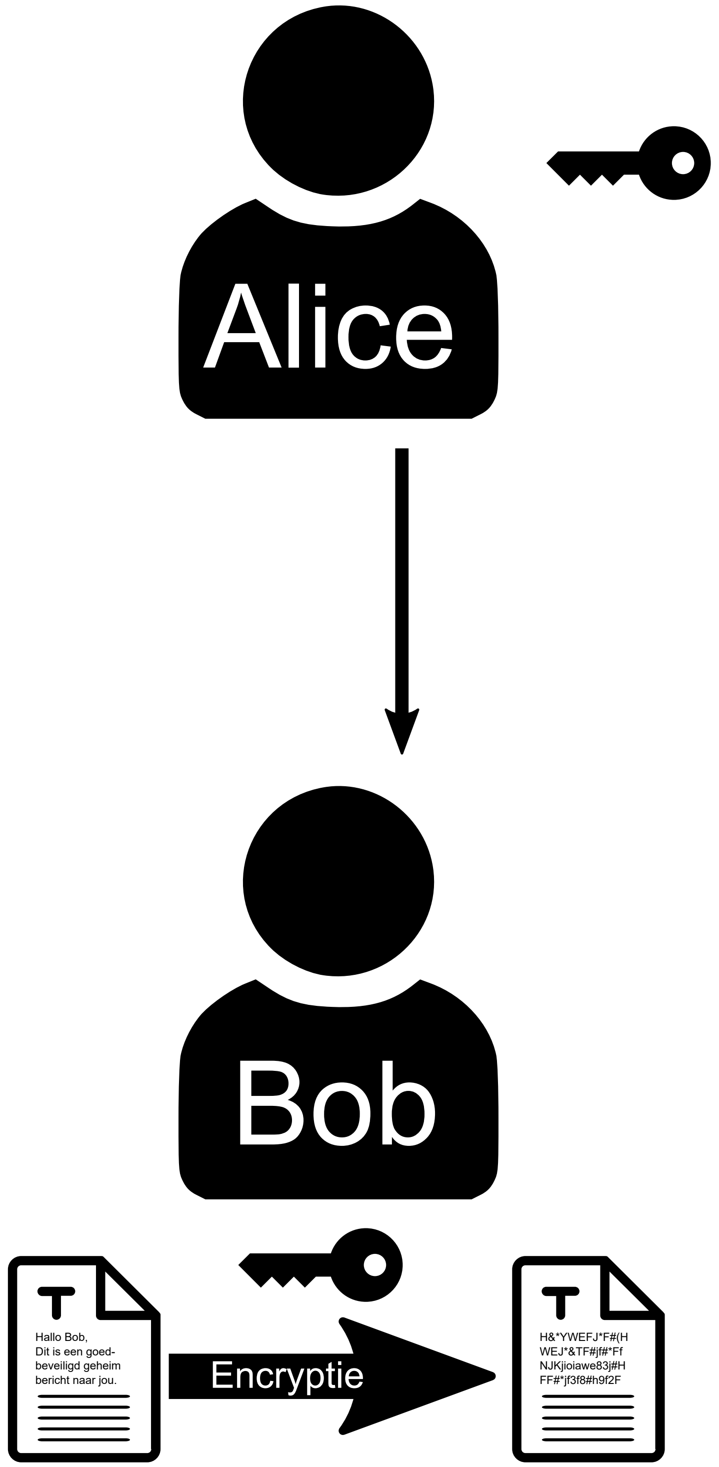
Hallo Bob,
Dit is een goed-beveiligd geheim bericht naar jou.

SYMMETRIC KEYS



Hallo Bob,
Dit is een goed-beveiligd geheim bericht naar jou.

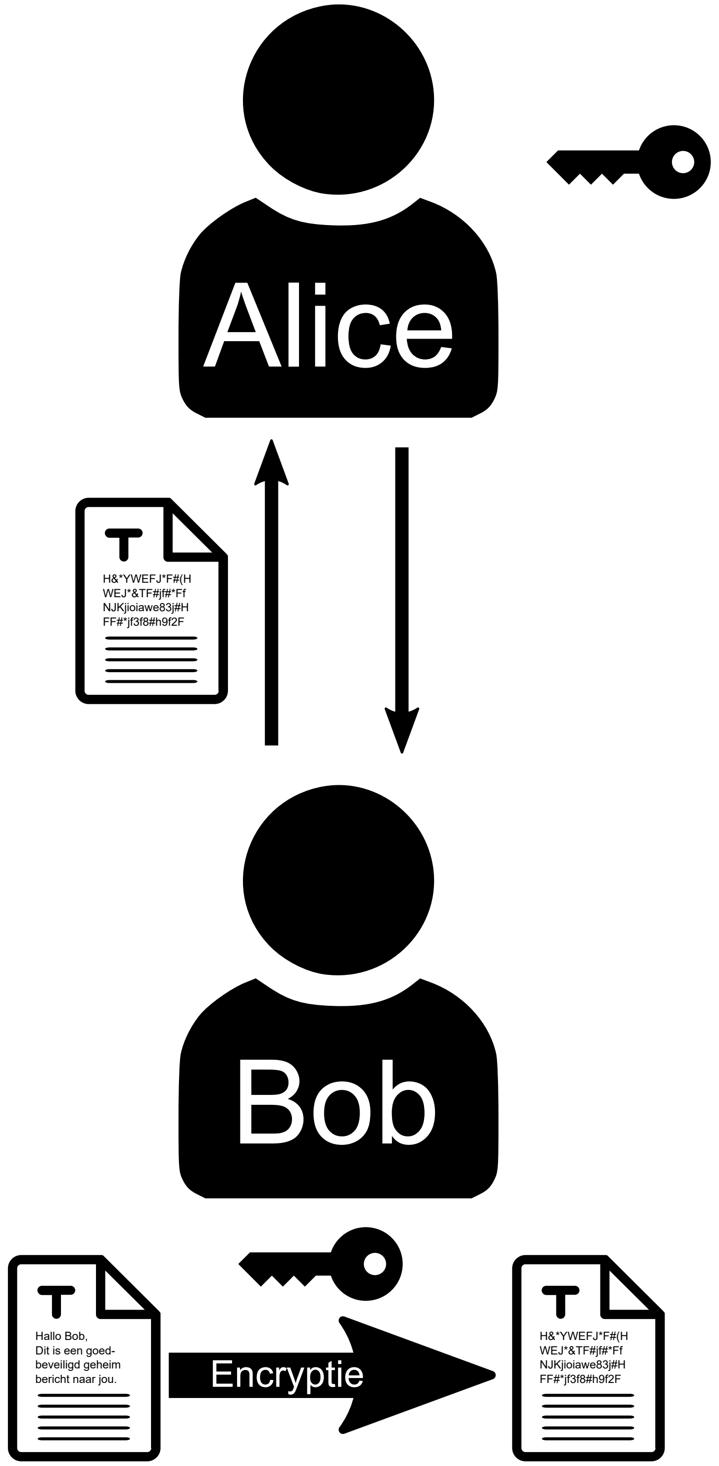
SYMMETRIC KEYS



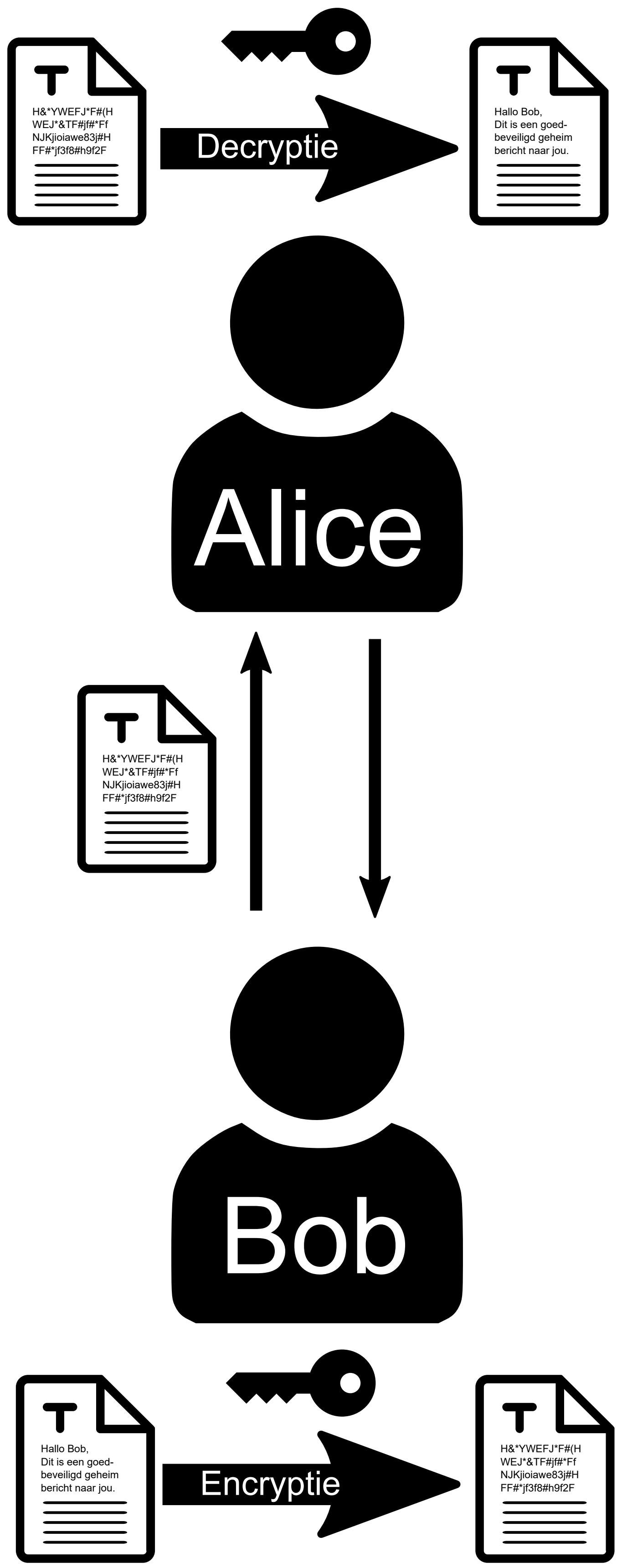
Hallo Bob,
Dit is een goed-beveiligd geheim bericht naar jou.

H&*YWEFJ*F#(H
WEJ*&TF#jf#*Ff
NJKjioiawe83j#H
FF#*jf3f8#h9f2F

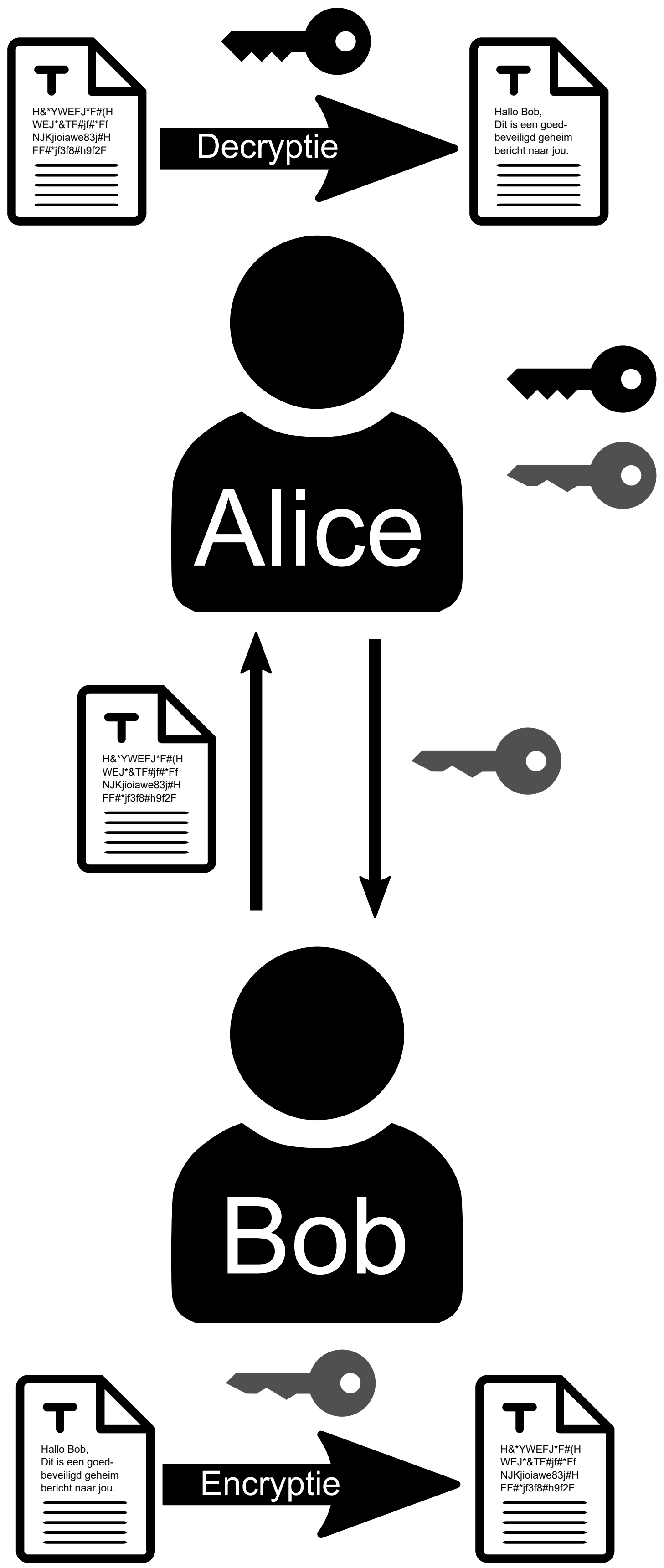
SYMMETRIC KEYS



SYMMETRIC KEYS



ASSYMETRIC KEYS



HTTPS HANDSHAKE

- de server deelt de public key en certificaat
- de client stuurt de session key, geencrypt met de public key van de server
- er wordt een encryptie methode bepaalt

HTTPS HANDSHAKE

- de server deelt de public key en certificaat
- de client stuurt de session key, geencrypt met de public key van de server
- er wordt een encryptie methode bepaalt

HTTPS HANDSHAKE

- de server deelt de public key en certificaat
- de client stuurt de session key, geencrypt met de public key van de server
- er wordt een encryptie methode bepaalt

🛡 HSTS

- HTTP Strict Transport Security
- Automatisch switchen naar HTTPS
  - door HTTP af te wijzen, of
  - door te redirecten
- Voorkomt man in de middle

🛡 HSTS

- HTTP Strict Transport Security
- Automatisch switchen naar HTTPS
  - door HTTP af te wijzen, of
  - door te redirecten
- Voorkomt man in de middle

🛡 HSTS

- HTTP Strict Transport Security
- Automatisch switchen naar HTTPS
  - door HTTP af te wijzen, of
  - door te redirecten
- Voorkomt man in de middle

🛡 HSTS

- HTTP Strict Transport Security
- Automatisch switchen naar HTTPS
  - door HTTP af te wijzen, of
  - door te redirecten
- Voorkomt man in de middle

🛡️ HSTS

- HTTP Strict Transport Security
- Automatisch switchen naar HTTPS
  - door HTTP af te wijzen, of
  - door te redirecten
- Voorkomt man in de middle

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

≡

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

≡

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

≡

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

  - Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
  - Alice logt in
  - Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

≡

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

  - Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
  - Alice logt in
  - Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

≡

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

≡

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

≡

⚔ SESSIEHIJACKING

- Herinner: HTTP is zelf stateless
- Na inloggen wordt er vaak 'sessie' gestart: met
  - ❌ sessie-id's als query-parameter
  - **cookies**
    - worden in de HTTP Response meegegeven als `Set-Cookie` (of in JS aangemaakt)
    - worden bij alle HTTP requests daarna meegestuurd
    - verlopen na een bepaalde tijd
    - met `HttpOnly` onbereikbaar voor JS (tegen XSS)
    - met `SameSite=Strict` worden de cookies van B.com niet meegestuurd als A.com een request doet naar B.com (tegen CSRF)

⚔Voorbeeld aanval (**session fixation**):

- Eve verleidt Alice om kwetsbaar.com/?session_id=123 te openen
- Alice logt in
- Eve heeft toegang tot Alice' account via kwetsbaar.com/?session_id=123

🛡Voorbeeld oplossing: na elke request/inlogpoging wordt de session_id ververst.

⚔Voorbeeld aanval (**session side jacking**): de cookie wordt gestolen op een open Wifi, en een website zonder HTTPS.

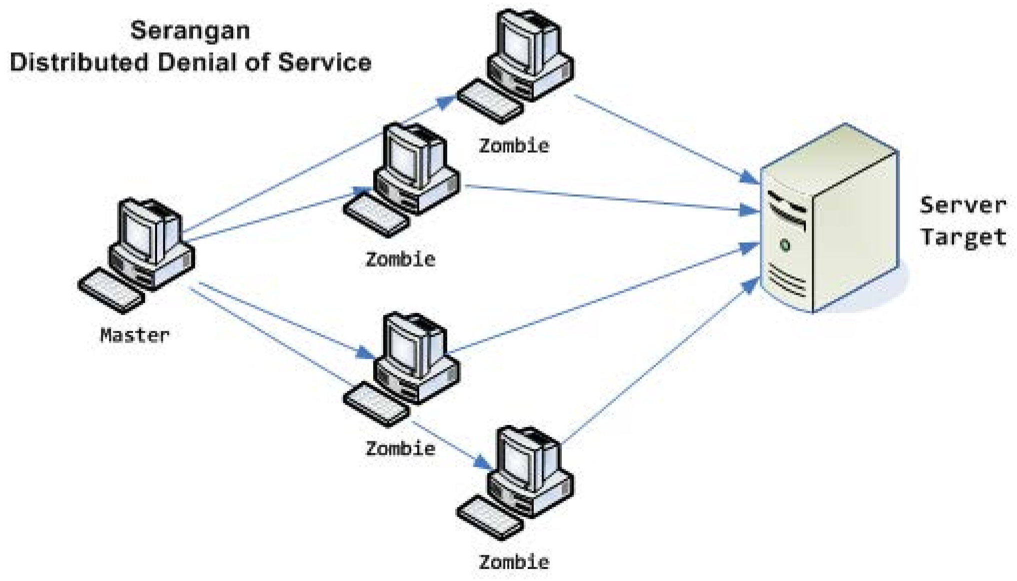
⚔ PHISHING

Kunnen wij niets aan doen tijdens de ontwikkeling van een webapplicatie

(Eventueel kunnen verdachte logins worden geblokeerd)

⚔ DDOS

Goede uitleg van Cloudflare hier.



Wat is het doel?

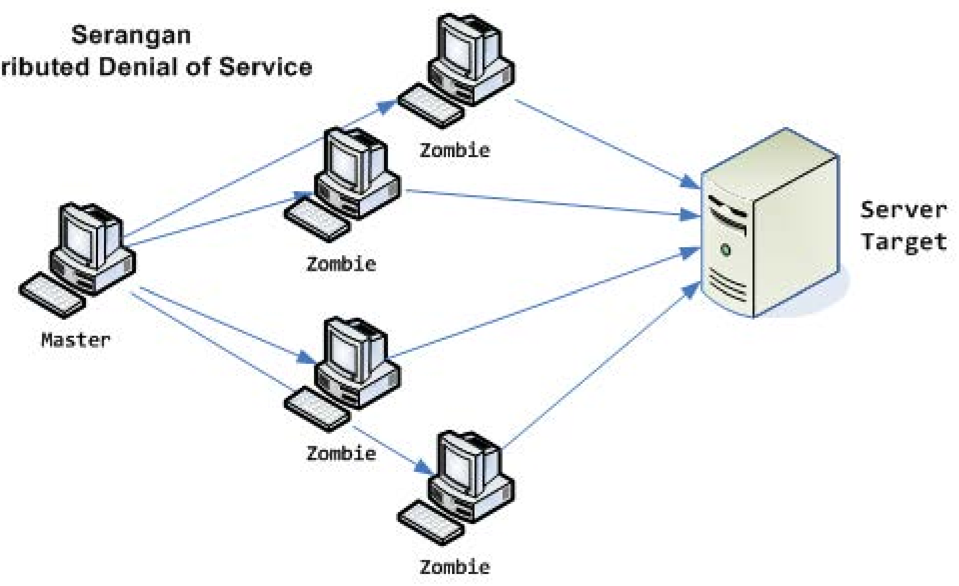
- Site plat leggen
- Afleiding voor een hack

Wat te doen? Bijvoorbeeld:

- Abnormale request(frequentie)
- DDOS challange: bijv. HTTP challenge
  - (bijv. redirect met cookies)
- IP reputation

⚔ DDOS

Goede uitleg van Cloudflare hier.



Wat is het doel?

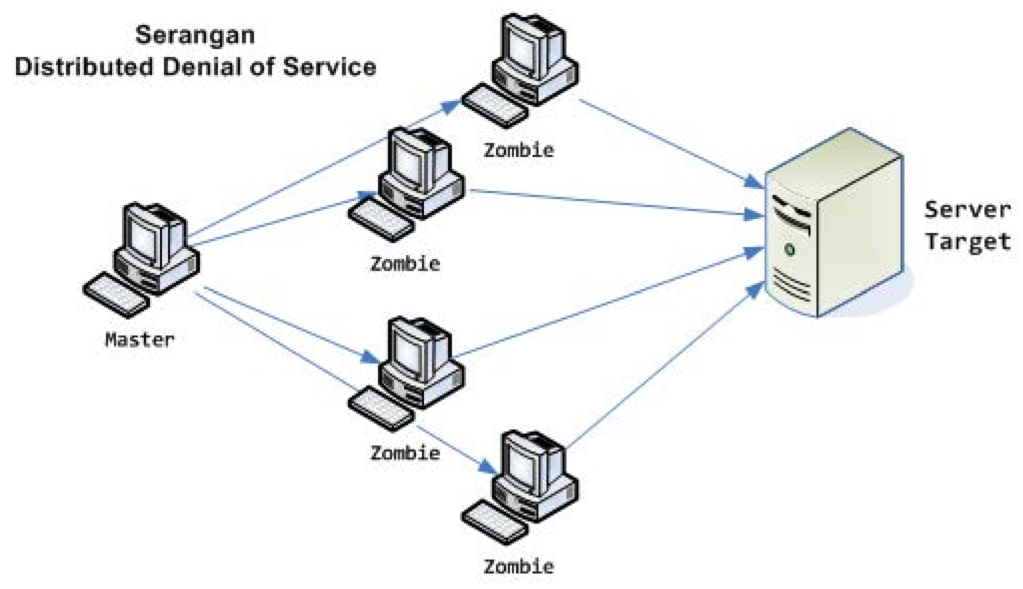
- Site plat leggen
- Afleiding voor een hack

Wat te doen? Bijvoorbeeld:

- Abnormale request(frequentie)
- DDOS challange: bijv. HTTP challenge
  - (bijv. redirect met cookies)
- IP reputation

⚔ DDOS

Goede uitleg van Cloudflare hier.



Wat is het doel?

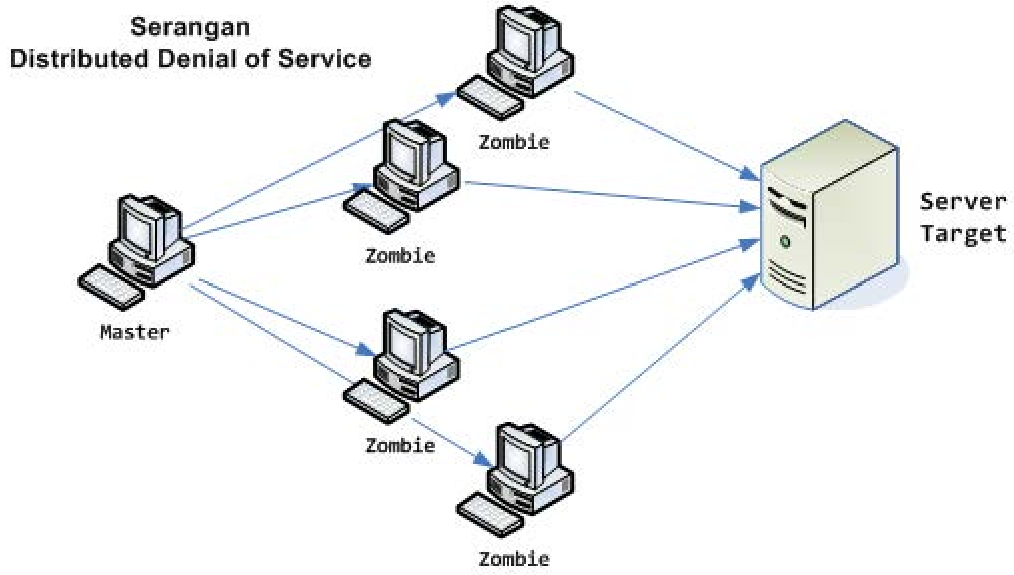
- Site plat leggen
- Afleiding voor een hack

Wat te doen? Bijvoorbeeld:

- Abnormale request(frequentie)
- DDOS challange: bijv. HTTP challenge
  - (bijv. redirect met cookies)
- IP reputation

⚔ DDOS

Goede uitleg van Cloudflare hier.



Wat is het doel?

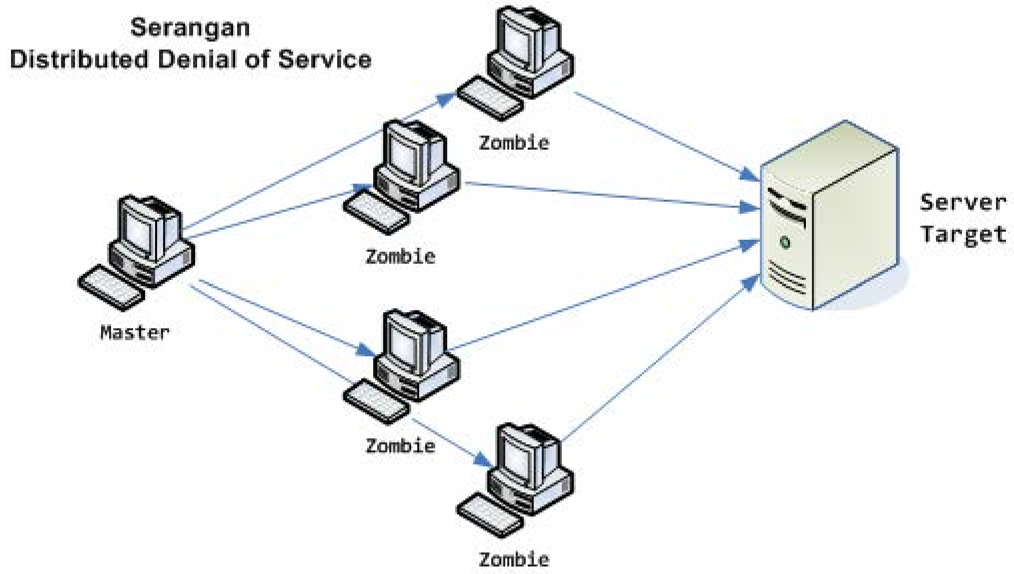
- Site plat leggen
- Afleiding voor een hack

Wat te doen? Bijvoorbeeld:

- Abnormale request(frequentie)
- DDOS challange: bijv. HTTP challenge
  - (bijv. redirect met cookies)
- IP reputation

⚔ DDOS

Goede uitleg van Cloudflare hier.



Wat is het doel?

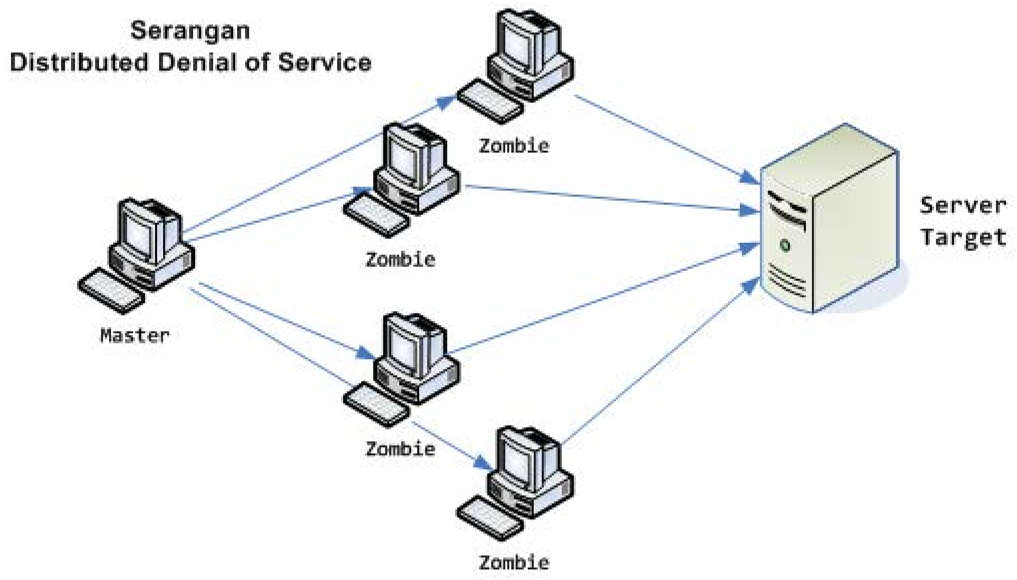
- Site plat leggen
- Afleiding voor een hack

Wat te doen? Bijvoorbeeld:

- Abnormale request(frequentie)
- DDOS challange: bijv. HTTP challenge
  - (bijv. redirect met cookies)
- IP reputation

⚔ DDOS

Goede uitleg van Cloudflare hier.



Wat is het doel?

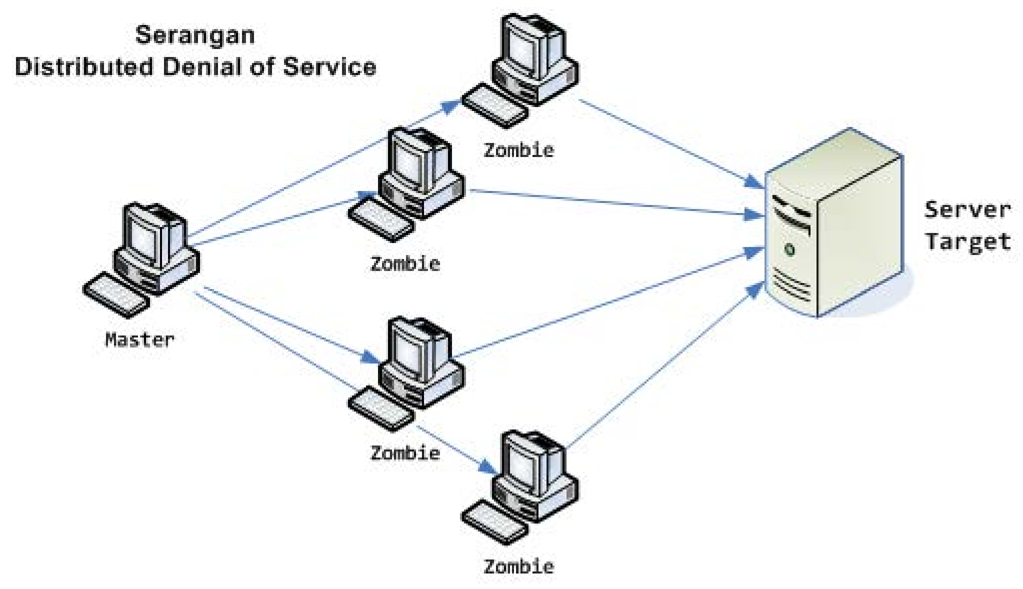
- Site plat leggen
- Afleiding voor een hack

Wat te doen? Bijvoorbeeld:

- Abnormale request(frequentie)
- DDOS challange: bijv. HTTP challenge
  - (bijv. redirect met cookies)
- IP reputation

⚔ DDOS

Goede uitleg van Cloudflare hier.



Wat is het doel?

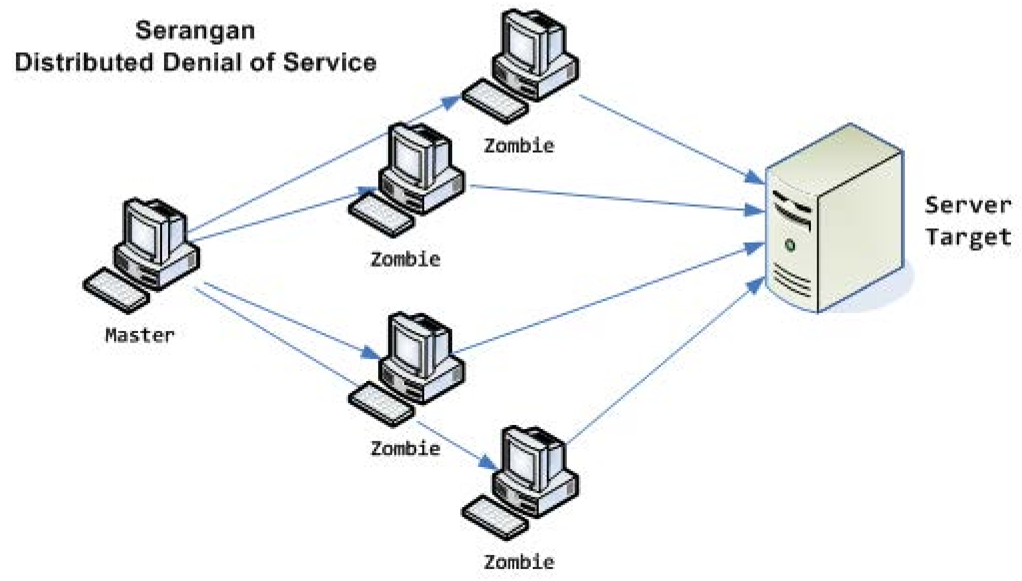
- Site plat leggen
- Afleiding voor een hack

Wat te doen? Bijvoorbeeld:

- Abnormale request(frequentie)
- DDOS challange: bijv. HTTP challenge
  - (bijv. redirect met cookies)
- IP reputation

⚔ DDOS

Goede uitleg van Cloudflare hier.



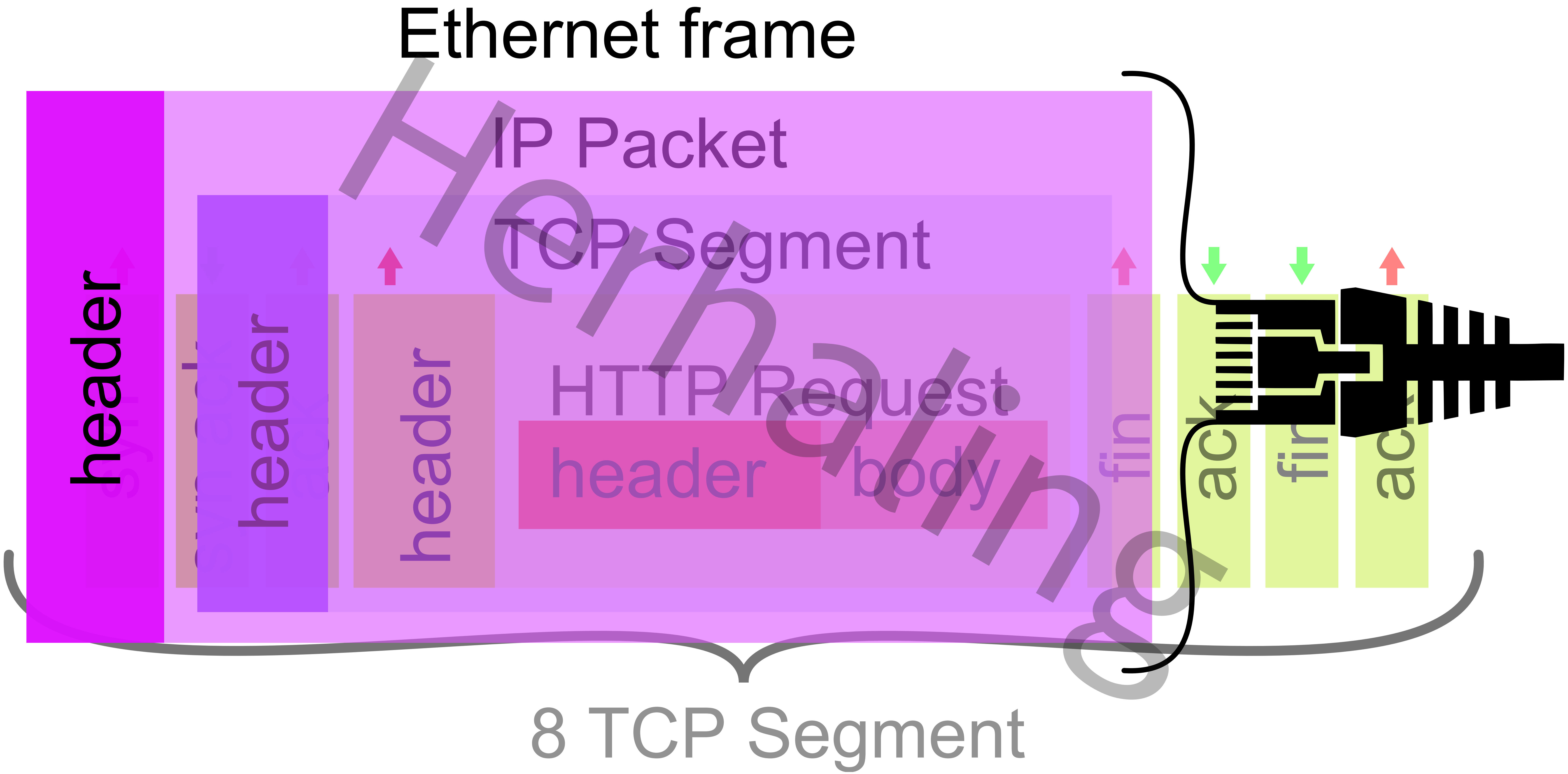
Wat is het doel?

- Site plat leggen
- Afleiding voor een hack

Wat te doen? Bijvoorbeeld:

- Abnormale request(frequentie)
- DDOS challange: bijv. HTTP challenge
    - (bijv. redirect met cookies)
- IP reputation

OSI LAGEN


Ethernet frame
IP Packet
TCP Segment
HTTP Request
header
header
header
header
body
syn
syn ack
ack
fin
ack
fin
ack
8 TCP Segment
Herhaling

DDOS IN ASP.NET?

DDOS aanval kan op elke laag in de netwerk communicatie (OSI model), maar is meestal op laag 4 of 7

- 2: Data link laag (ethernet frame, ...)
- 3: Netwerk laag (IP packet, ...)
- 4: Transport laag (TCP, UDP, ...): bijvoorbeeld SYN floods
- 7: Applicatie laag (HTTP)

Voor laag 1 tot 6 is er in de ASP.NET controllers is er niets aan DDOS te doen.

De instellingen van de server (IIS) kunnen wel helpen (vooral voor laag 7).

Packages kunnen helpen: zoals deze.

DDOS IN ASP.NET?

DDOS aanval kan op elke laag in de netwerk communicatie (OSI model), maar is meestal op laag 4 of 7

- 2: Data link laag (ethernet frame, ...)
- 3: Netwerk laag (IP packet, ...)
- 4: Transport laag (TCP, UDP, ...): bijvoorbeeld SYN floods
- 7: Applicatie laag (HTTP)

Voor laag 1 tot 6 is er in de ASP.NET controllers is er niets aan DDOS te doen.

De instellingen van de server (IIS) kunnen wel helpen (vooral voor laag 7).

Packages kunnen helpen: zoals deze.

DDOS IN ASP.NET?

DDOS aanval kan op elke laag in de netwerk communicatie (OSI model), maar is meestal op laag 4 of 7

- 2: Data link laag (ethernet frame, ...)
- 3: Netwerk laag (IP packet, ...)
- 4: Transport laag (TCP, UDP, ...): bijvoorbeeld SYN floods
- 7: Applicatie laag (HTTP)

Voor laag 1 tot 6 is er in de ASP.NET controllers is er niets aan DDOS te doen.

De instellingen van de server (IIS) kunnen wel helpen (vooral voor laag 7).

Packages kunnen helpen: zoals deze.

DDOS IN ASP.NET?

DDOS aanval kan op elke laag in de netwerk communicatie (OSI model), maar is meestal op laag 4 of 7

- 2: Data link laag (ethernet frame, ...)
- 3: Netwerk laag (IP packet, ...)
- 4: Transport laag (TCP, UDP, ...): bijvoorbeeld SYN floods
- 7: Applicatie laag (HTTP)

Voor laag 1 tot 6 is er in de ASP.NET controllers is er niets aan DDOS te doen.

De instellingen van de server (IIS) kunnen wel helpen (vooral voor laag 7).

Packages kunnen helpen: zoals deze.

🛡 DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

🛡 DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

🛡 DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

🛡️ DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

🛡️ DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

🛡 DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

🛡 DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

🛡 DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

🛡 DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

🛡️ DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

🛡 DATABASE BEVEILIGING

- OWASP principe *Defence in depth*: beveiliging op database-niveau.
- Verschillende database gebruikers:
  - Administrators
  - Data analysten
  - Politie
  - Reclame bureaus
- Hoe?
  - Database gebruikers/rollen
  - Encryptie van de database (database level, collumn level, ...)
  - Views
  - Pseudonimisatie

VIEWS IN EF CORE

- Gegeven een `MijnContext` met een `Student`:

```
[Keyless]
public class AnoniemeStudent
{
    public int Id { get; set; }
    public double Cijfer { get; set; }
}
```

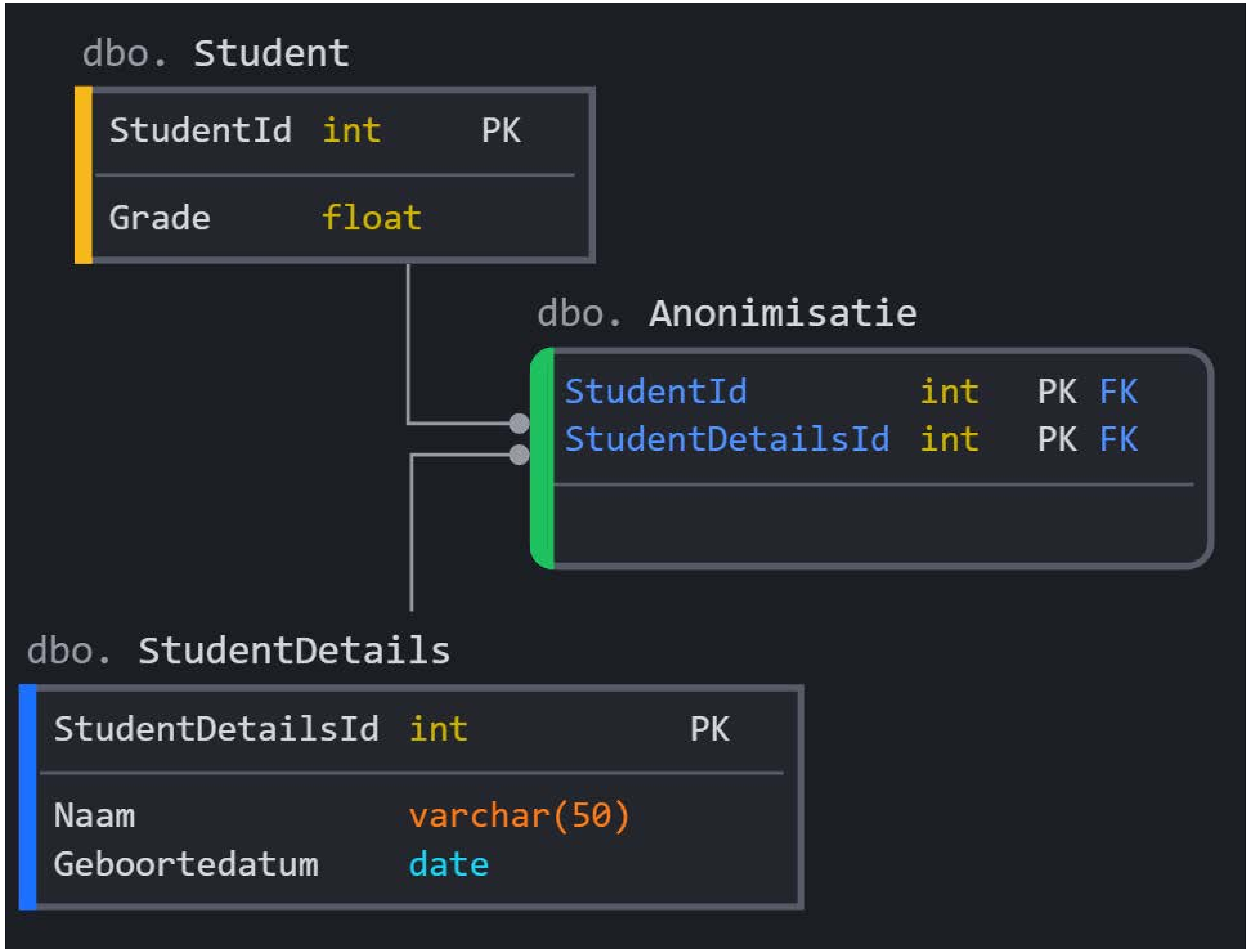
- View aanmaken in de database:

```
public partial class editUserDTO : DbMigration
{
    public override void Up()
    {
        MijnContext ctx = new MijnContext();
        ctx.Database.ExecuteSqlCommand(
            @"CREATE VIEW dbo.AnoniemeStudent
            AS SELECT ID, Cijfer
            FROM dbo.Student"
        );
    }
    public override void Down()
    {
        MijnContext ctx = new MijnContext();
        ctx.Database.ExecuteSqlCommand("DROP VIEW dbo.AnoniemeStudent");
    }
}
```

- Het bijbehorende model:

```
[Keyless]
public class AnoniemeStudent
{
    public int Id { get; set; }
    public double Cijfer { get; set; }
}
```

PSEUDONIMISATIE



```
dbo. Student
StudentId   int      PK
Grade       float

dbo. Anonimisatie
StudentId          int    PK FK
StudentDetailsId   int    PK FK

dbo. StudentDetails
StudentDetailsId   int            PK
Naam               varchar(50)
Geboortedatum      date
```

🛡 ZET DE DEVELOPER HULPMIDDELEN UIT

- Geen zinvolle errors:
  - Geen `app.UseDeveloperExceptionPage();`
  - Geen `app.UseDatabaseErrorPage();`
- Log zo veel mogelijk
  - Set de log-levels in `appsettings.json`
  - Gebruik de `ILogger<StudentController> logger` dependancy injection

🛡 ZET DE DEVELOPER HULPMIDDELEN UIT

- Geen zinvolle errors:
  - Geen `app.UseDeveloperExceptionPage();`
  - Geen `app.UseDatabaseErrorPage();`
- Log zo veel mogelijk
  - Set de log-levels in `appsettings.json`
  - Gebruik de `ILogger<StudentController> logger` dependancy injection

☰

🛡 ZET DE DEVELOPER HULPMIDDELEN UIT

- Geen zinvolle errors:
  - Geen `app.UseDeveloperExceptionPage();`
  - Geen `app.UseDatabaseErrorPage();`
- Log zo veel mogelijk
  - Set de log-levels in `appsettings.json`
  - Gebruik de `ILogger<StudentController> logger` dependancy injection

🛡 ZET DE DEVELOPER HULPMIDDELEN UIT

- Geen zinvolle errors:
  - Geen `app.UseDeveloperExceptionPage();`
  - Geen `app.UseDatabaseErrorPage();`
- Log zo veel mogelijk
  - Set de log-levels in `appsettings.json`
  - Gebruik de `ILogger<StudentController> logger` dependancy injection

🛡 ZET DE DEVELOPER HULPMIDDELEN UIT

- Geen zinvolle errors:
  - Geen `app.UseDeveloperExceptionPage();`
  - Geen `app.UseDatabaseErrorPage();`
- Log zo veel mogelijk
  - Set de log-levels in `appsettings.json`
  - Gebruik de `ILogger<StudentController> logger` dependancy injection

🛡 ZET DE DEVELOPER HULPMIDDELEN UIT

- Geen zinvolle errors:
  - Geen `app.UseDeveloperExceptionPage();`
  - Geen `app.UseDatabaseErrorPage();`
- Log zo veel mogelijk
  - Set de log-levels in `appsettings.json`
  - Gebruik de `ILogger<StudentController> logger` dependancy injection

🛡️ ZET DE DEVELOPER HULPMIDDELEN UIT

- Geen zinvolle errors:
  - Geen `app.UseDeveloperExceptionPage();`
  - Geen `app.UseDatabaseErrorPage();`
- Log zo veel mogelijk
  - Set de log-levels in `appsettings.json`
  - Gebruik de `ILogger<StudentController> logger` dependancy injection

🛡 AUTHENTICATIE EN AUTHORISATIE

- Stel: je wilt je e-mail (bijvoorbeeld gmail) lezen in je browser
- Of: je wilt je hotelboeking wijzigen of annuleren
- Dit zijn acties die jij graag wilt uitvoeren maar het is niet de bedoeling dat anderen dat ook kunnen (zien)
- Daarvoor gebruiken we een heel bekende oplossing: inloggen met een wachtwoord
- ASP.NET Core 3.1 kent die mogelijkheid natuurlijk ook, maar je moet wel even het juiste vinkje zetten

🛡 AUTHENTICATIE EN AUTHORISATIE

- Stel: je wilt je e-mail (bijvoorbeeld gmail) lezen in je browser
- Of: je wilt je hotelboeking wijzigen of annuleren
- Dit zijn acties die jij graag wilt uitvoeren maar het is niet de bedoeling dat anderen dat ook kunnen (zien)
- Daarvoor gebruiken we een heel bekende oplossing: inloggen met een wachtwoord
- ASP.NET Core 3.1 kent die mogelijkheid natuurlijk ook, maar je moet wel even het juiste vinkje zetten

🛡 AUTHENTICATIE EN AUTHORISATIE

- Stel: je wilt je e-mail (bijvoorbeeld gmail) lezen in je browser
- Of: je wilt je hotelboeking wijzigen of annuleren
- Dit zijn acties die jij graag wilt uitvoeren maar het is niet de bedoeling dat anderen dat ook kunnen (zien)
- Daarvoor gebruiken we een heel bekende oplossing: inloggen met een wachtwoord
- ASP.NET Core 3.1 kent die mogelijkheid natuurlijk ook, maar je moet wel even het juiste vinkje zetten

🛡 AUTHENTICATIE EN AUTHORISATIE

- Stel: je wilt je e-mail (bijvoorbeeld gmail) lezen in je browser
- Of: je wilt je hotelboeking wijzigen of annuleren
- Dit zijn acties die jij graag wilt uitvoeren maar het is niet de bedoeling dat anderen dat ook kunnen (zien)
- Daarvoor gebruiken we een heel bekende oplossing: inloggen met een wachtwoord
- ASP.NET Core 3.1 kent die mogelijkheid natuurlijk ook, maar je moet wel even het juiste vinkje zetten

🛡️ AUTHENTICATIE EN AUTHORISATIE

- Stel: je wilt je e-mail (bijvoorbeeld gmail) lezen in je browser
- Of: je wilt je hotelboeking wijzigen of annuleren
- Dit zijn acties die jij graag wilt uitvoeren maar het is niet de bedoeling dat anderen dat ook kunnen (zien)
- Daarvoor gebruiken we een heel bekende oplossing: inloggen met een wachtwoord
- ASP.NET Core 3.1 kent die mogelijkheid natuurlijk ook, maar je moet wel even het juiste vinkje zetten

BEGRIPPEN

- **Identificatie:** vaststellen wie een persoon is.
- **Authenticatie:** Aantonen dat de vastgestelde identiteit correct is.
  - Wie ben jij?
  - Authenticatie kan door een derde partij (OAuth): geen extra wachtwoord!
- **Toegangscontrole** (access control): Beperking van wie op welke manier wat kan doen
- **Autorisatie:** Het proces dat vaststelt wat een gebruiker wel en niet kan en mag met de applicatie (rollen/rechten/privileges)
  - Wat mag jij?

BEGRIPPEN

- **Identificatie**: vaststellen wie een persoon is.
- **Authenticatie**: Aantonen dat de vastgestelde identiteit correct is.
    - Wie ben jij?
    - Authenticatie kan door een derde partij (OAuth): geen extra wachtwoord!
- **Toegangscontrole** (access control): Beperking van wie op welke manier wat kan doen
- **Autorisatie**: Het proces dat vaststelt wat een gebruiker wel en niet kan en mag met de applicatie (rollen/rechten/privileges)
    - Wat mag jij?

BEGRIPPEN

- **Identificatie**: vaststellen wie een persoon is.
- **Authenticatie**: Aantonen dat de vastgestelde identiteit correct is.
  - Wie ben jij?
  - Authenticatie kan door een derde partij (OAuth): geen extra wachtwoord!
- **Toegangscontrole** (access control): Beperking van wie op welke manier wat kan doen
- **Autorisatie**: Het proces dat vaststelt wat een gebruiker wel en niet kan en mag met de applicatie (rollen/rechten/privileges)
  - Wat mag jij?

BEGRIPPEN

- **Identificatie**: vaststellen wie een persoon is.
- **Authenticatie**: Aantonen dat de vastgestelde identiteit correct is.
  - Wie ben jij?
  - Authenticatie kan door een derde partij (OAuth): geen extra wachtwoord!
- **Toegangscontrole** (access control): Beperking van wie op welke manier wat kan doen
- **Autorisatie**: Het proces dat vaststelt wat een gebruiker wel en niet kan en mag met de applicatie (rollen/rechten/privileges)
  - Wat mag jij?

≡

BEGRIPPEN

- **Identificatie**: vaststellen wie een persoon is.
- **Authenticatie**: Aantonen dat de vastgestelde identiteit correct is.
  - Wie ben jij?
  - Authenticatie kan door een derde partij (OAuth): geen extra wachtwoord!
- **Toegangscontrole** (access control): Beperking van wie op welke manier wat kan doen
- **Autorisatie**: Het proces dat vaststelt wat een gebruiker wel en niet kan en mag met de applicatie (rollen/rechten/privileges)
  - Wat mag jij?

BEGRIPPEN

- **Identificatie**: vaststellen wie een persoon is.
- **Authenticatie**: Aantonen dat de vastgestelde identiteit correct is.
  - Wie ben jij?
  - Authenticatie kan door een derde partij (OAuth): geen extra wachtwoord!
- **Toegangscontrole** (access control): Beperking van wie op welke manier wat kan doen
- **Autorisatie**: Het proces dat vaststelt wat een gebruiker wel en niet kan en mag met de applicatie (rollen/rechten/privileges)
  - Wat mag jij?

BEGRIPPEN

- **Identificatie**: vaststellen wie een persoon is.
- **Authenticatie**: Aantonen dat de vastgestelde identiteit correct is.
  - Wie ben jij?
  - Authenticatie kan door een derde partij (OAuth): geen extra wachtwoord!
- **Toegangscontrole** (access control): Beperking van wie op welke manier wat kan doen
- **Autorisatie**: Het proces dat vaststelt wat een gebruiker wel en niet kan en mag met de applicatie (rollen/rechten/privileges)
  - Wat mag jij?

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

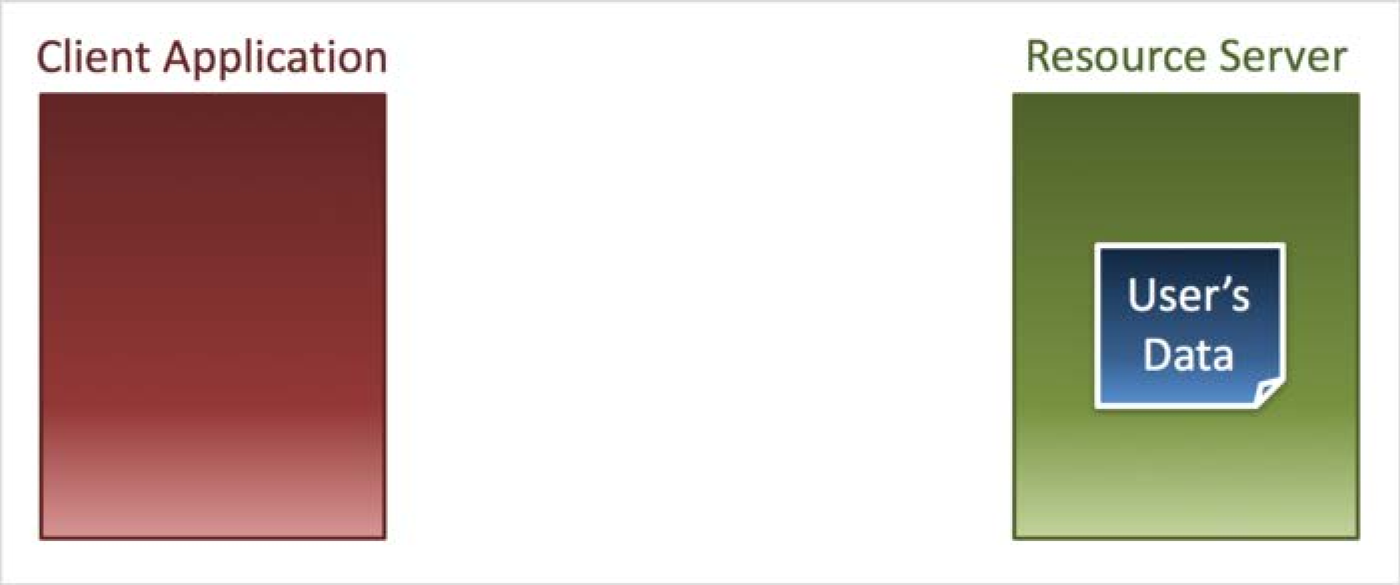
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).

Resource Server
User's Data

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

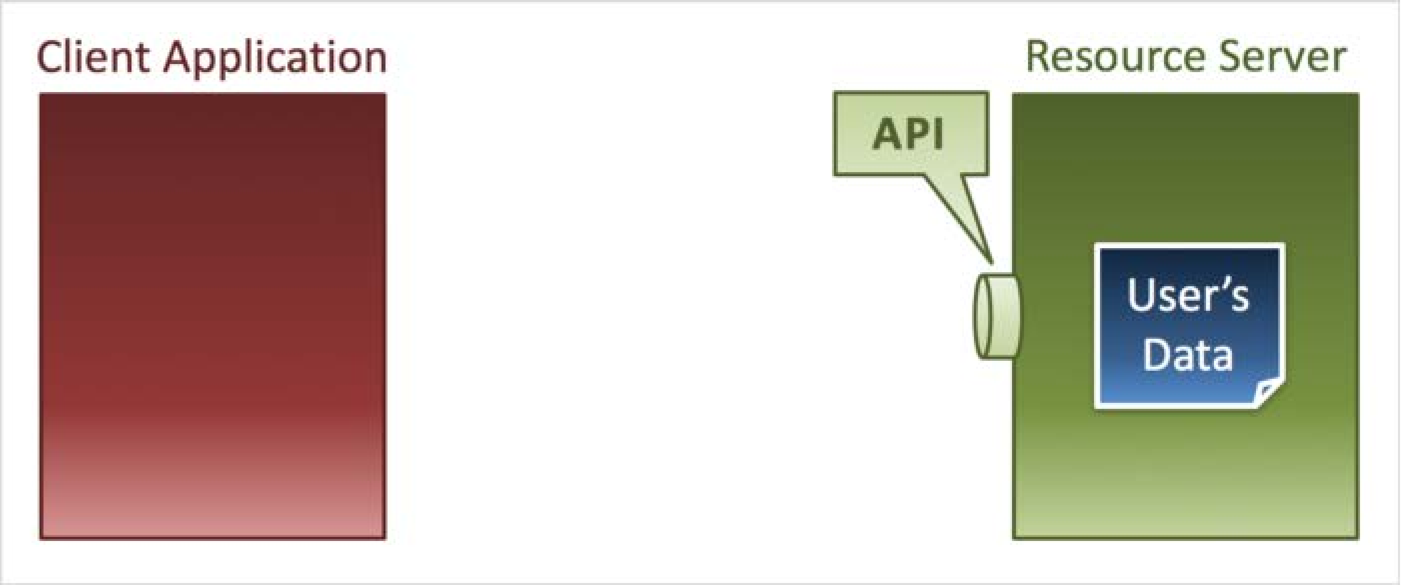
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

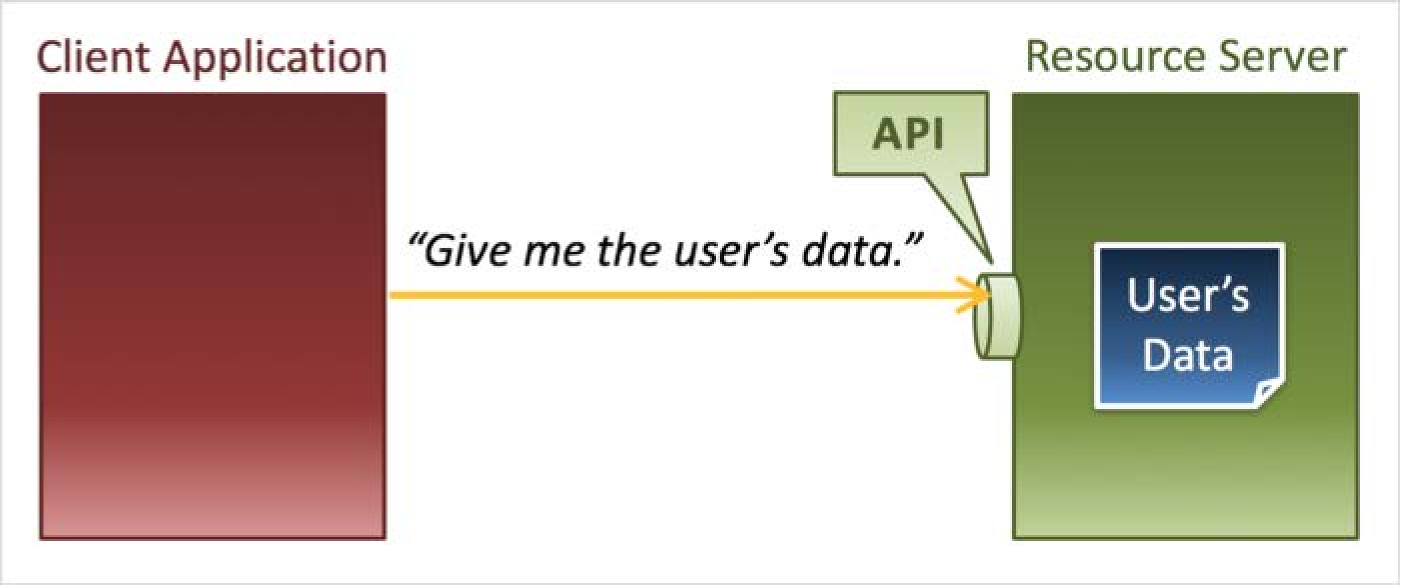
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

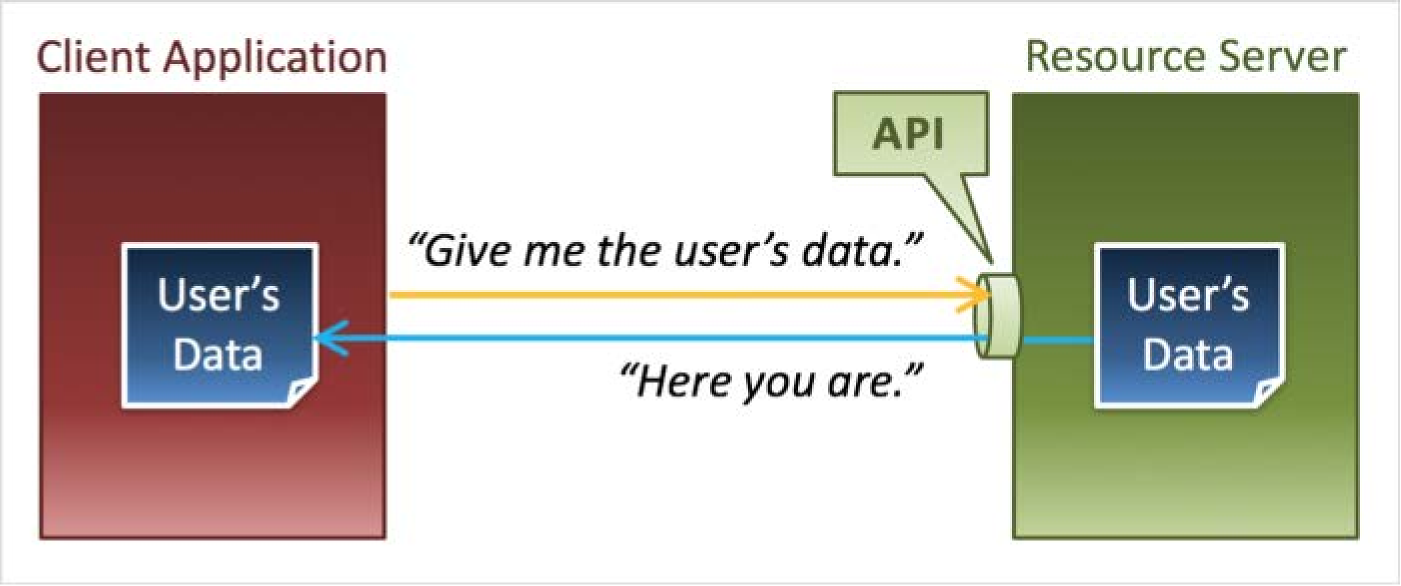
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

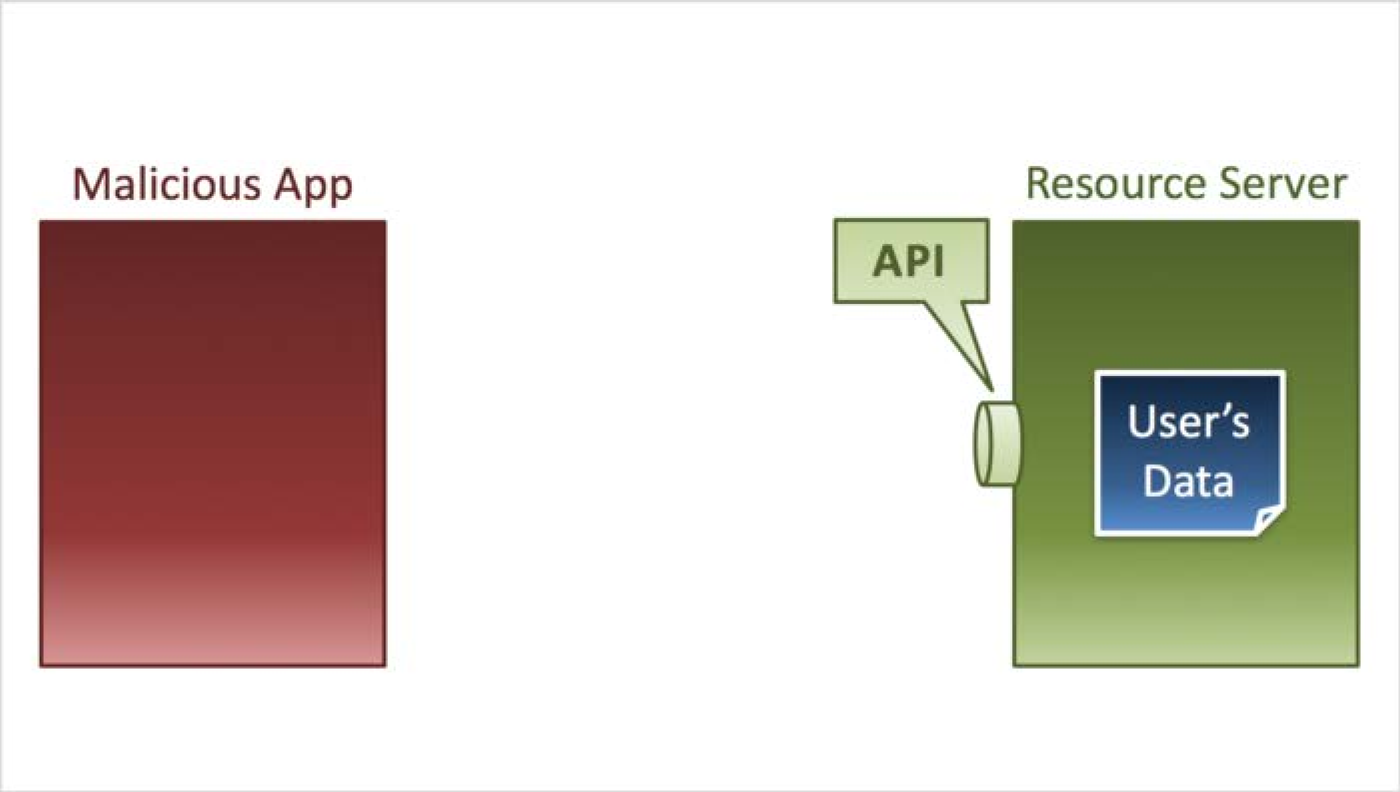
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

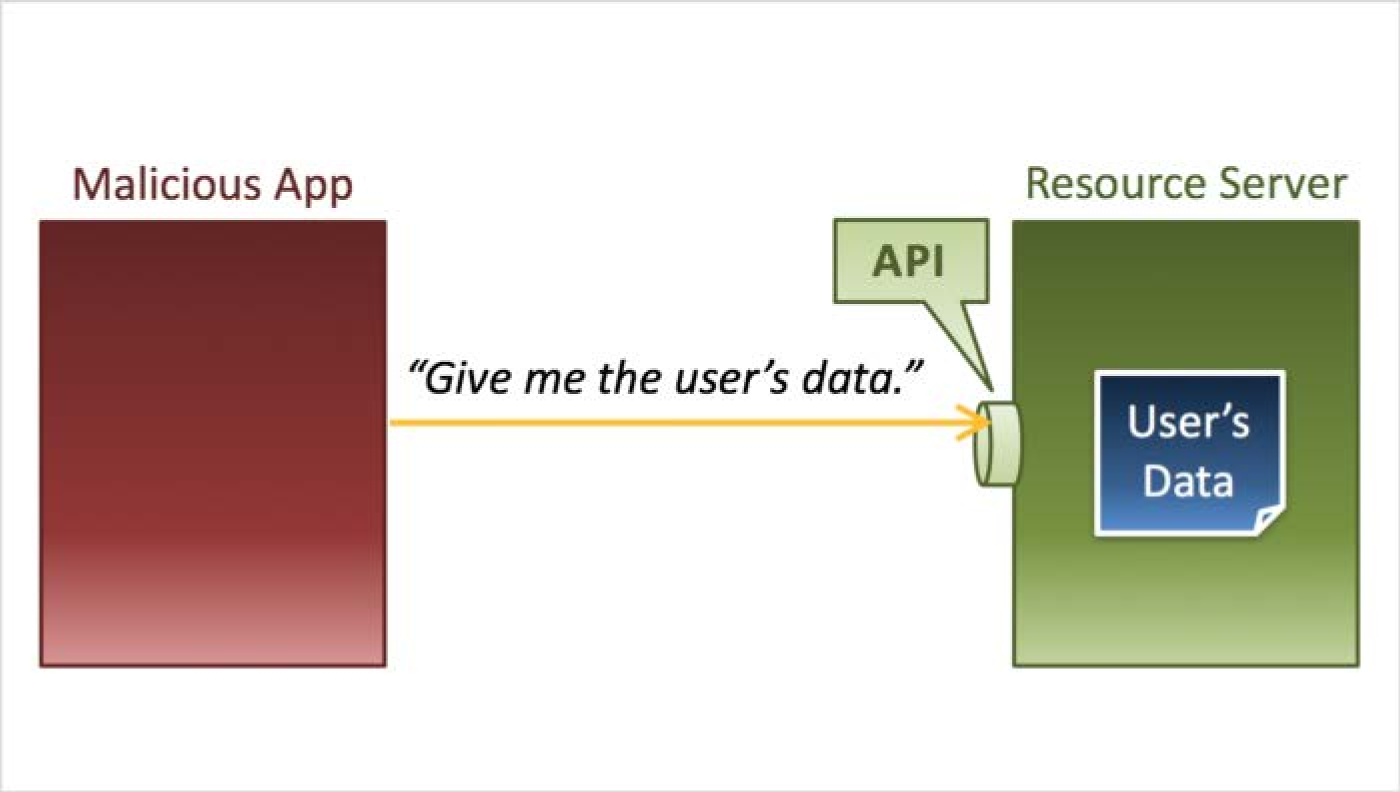
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

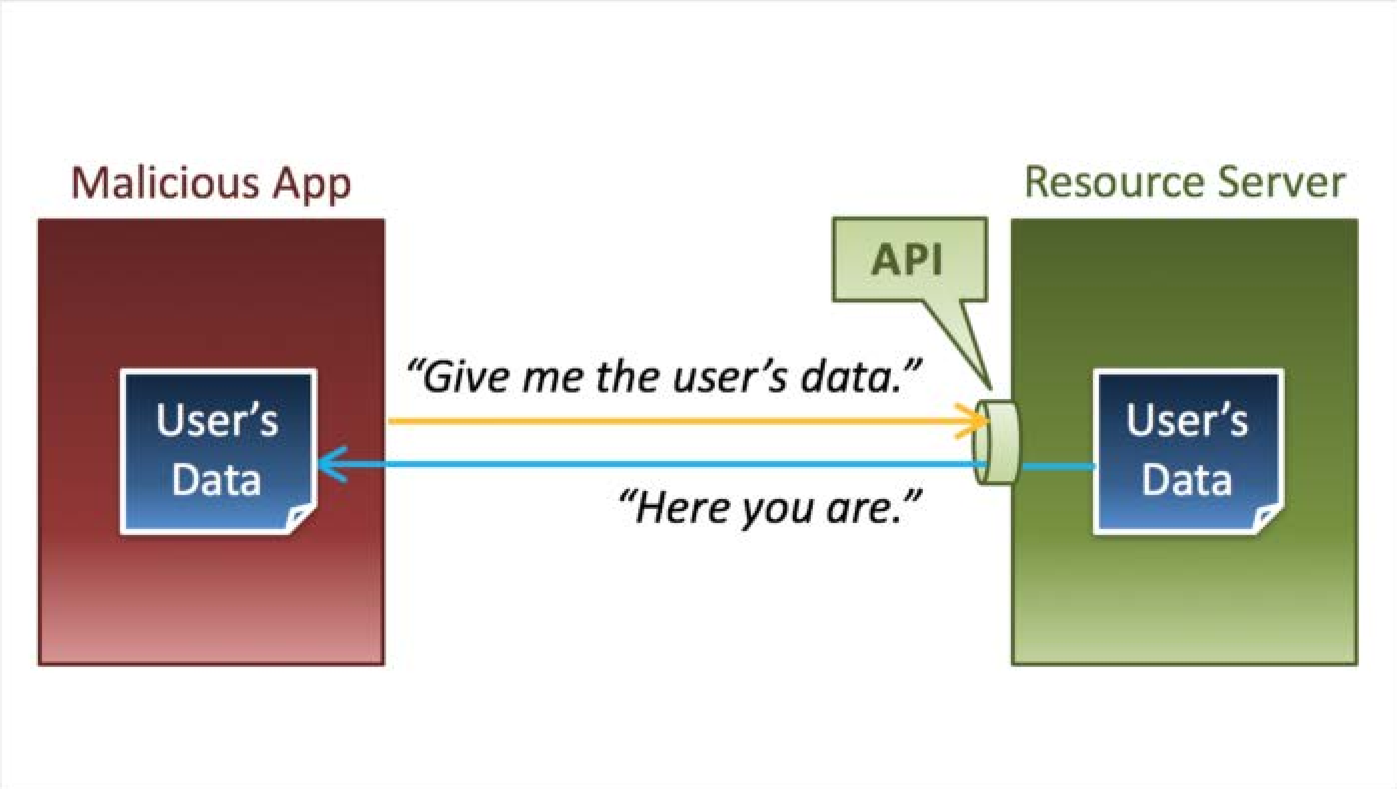
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

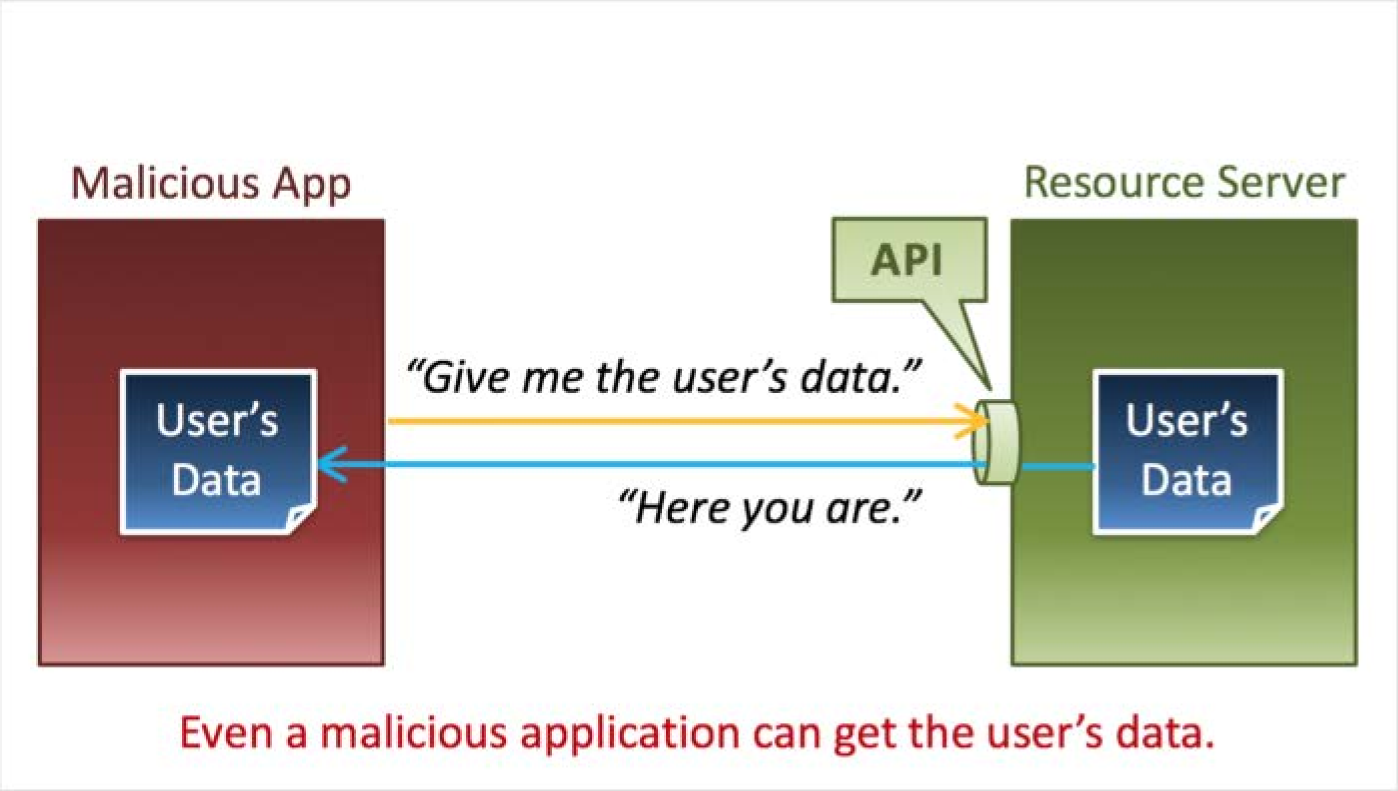
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

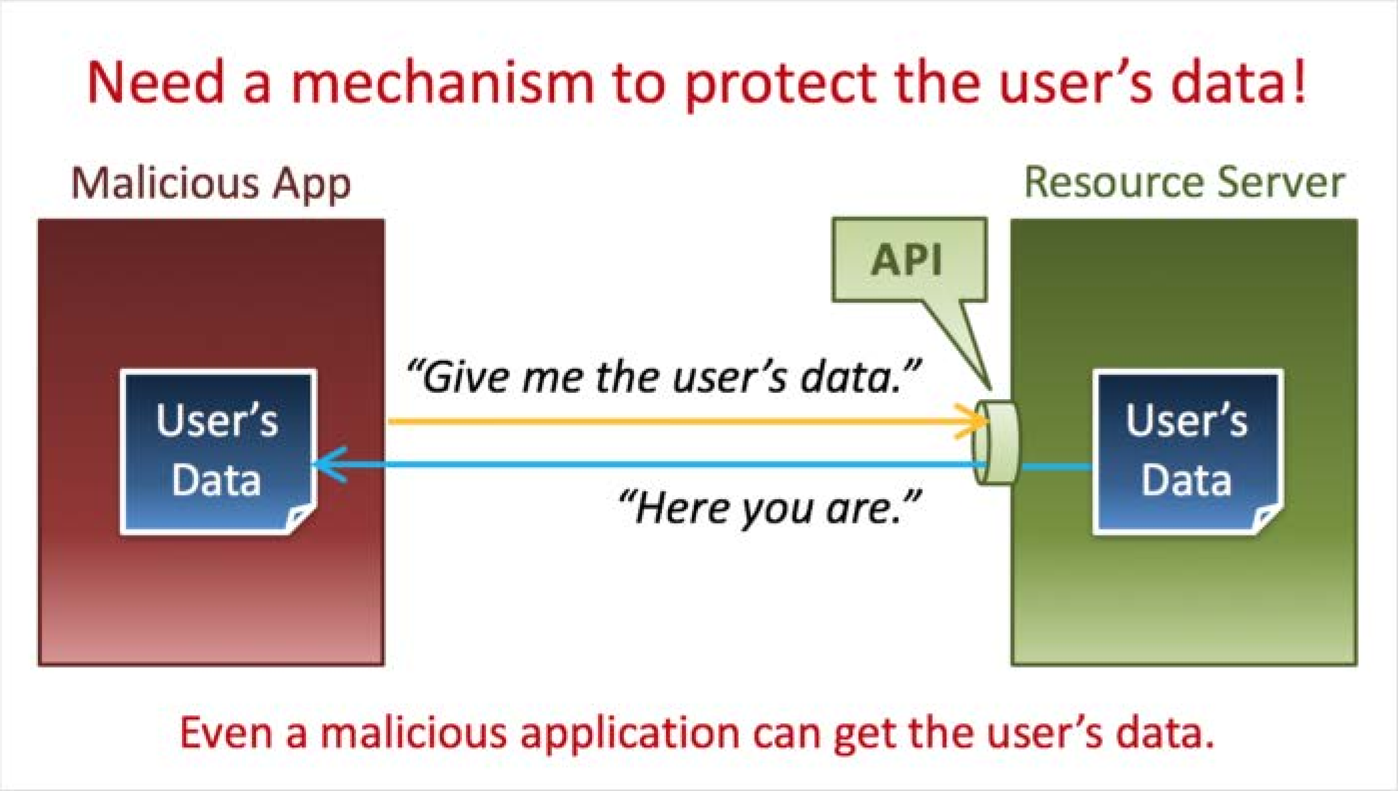
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).


Malicious App
Resource Server
API
User's Data
"Give me the user's data."
"Here you are."
User's Data
Even a malicious application can get the user's data.

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

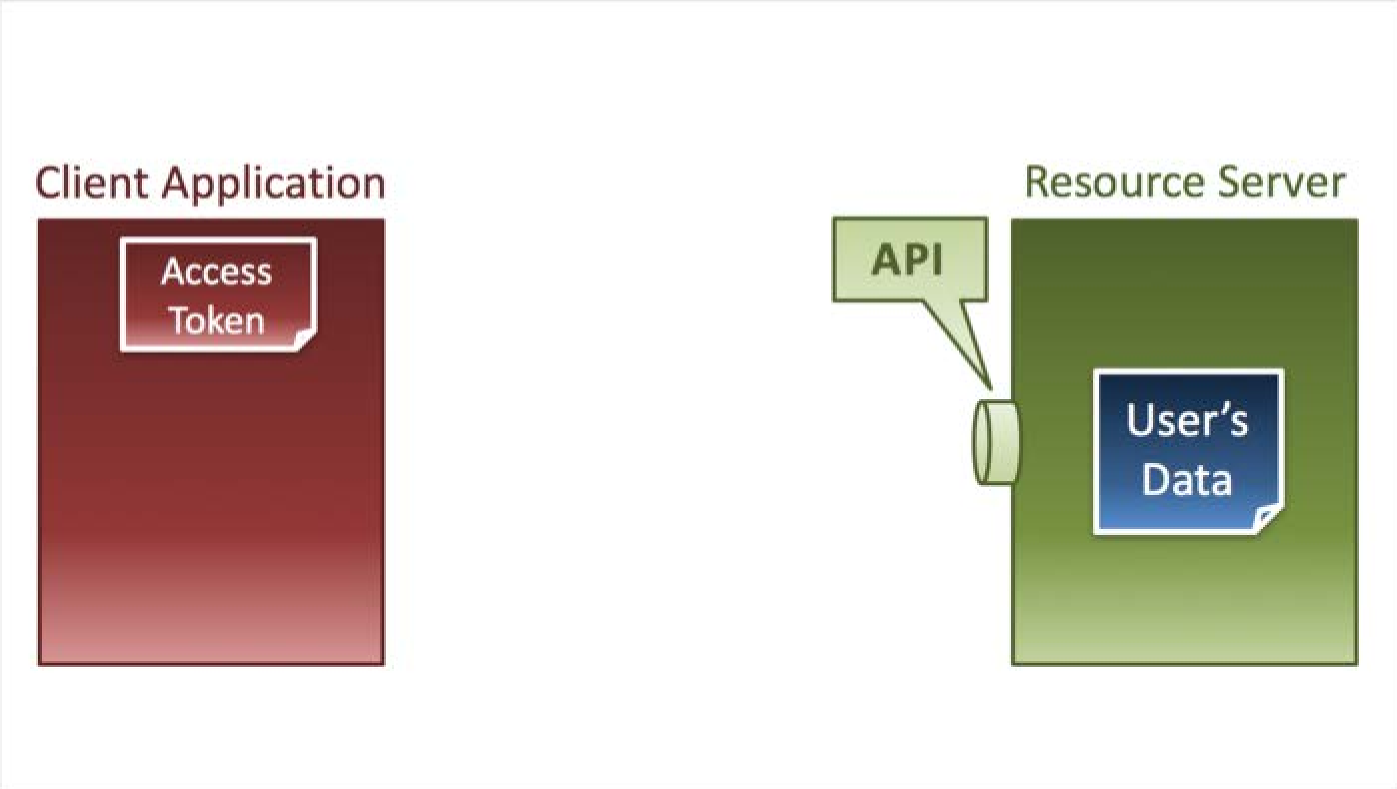
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).


Need a mechanism to protect the user's data!
Malicious App
Resource Server
API
User's Data
"Give me the user's data."
"Here you are."
User's Data
Even a malicious application can get the user's data.

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

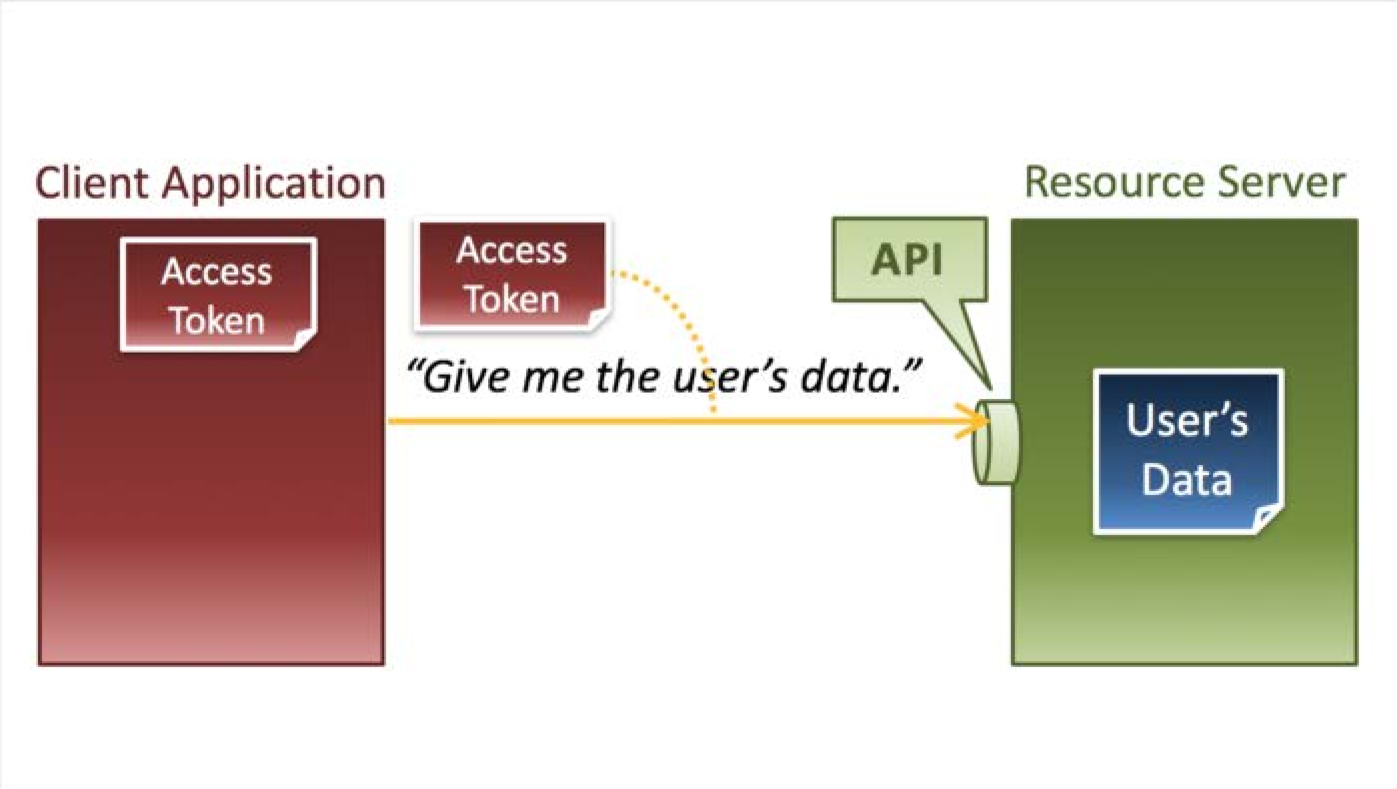
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

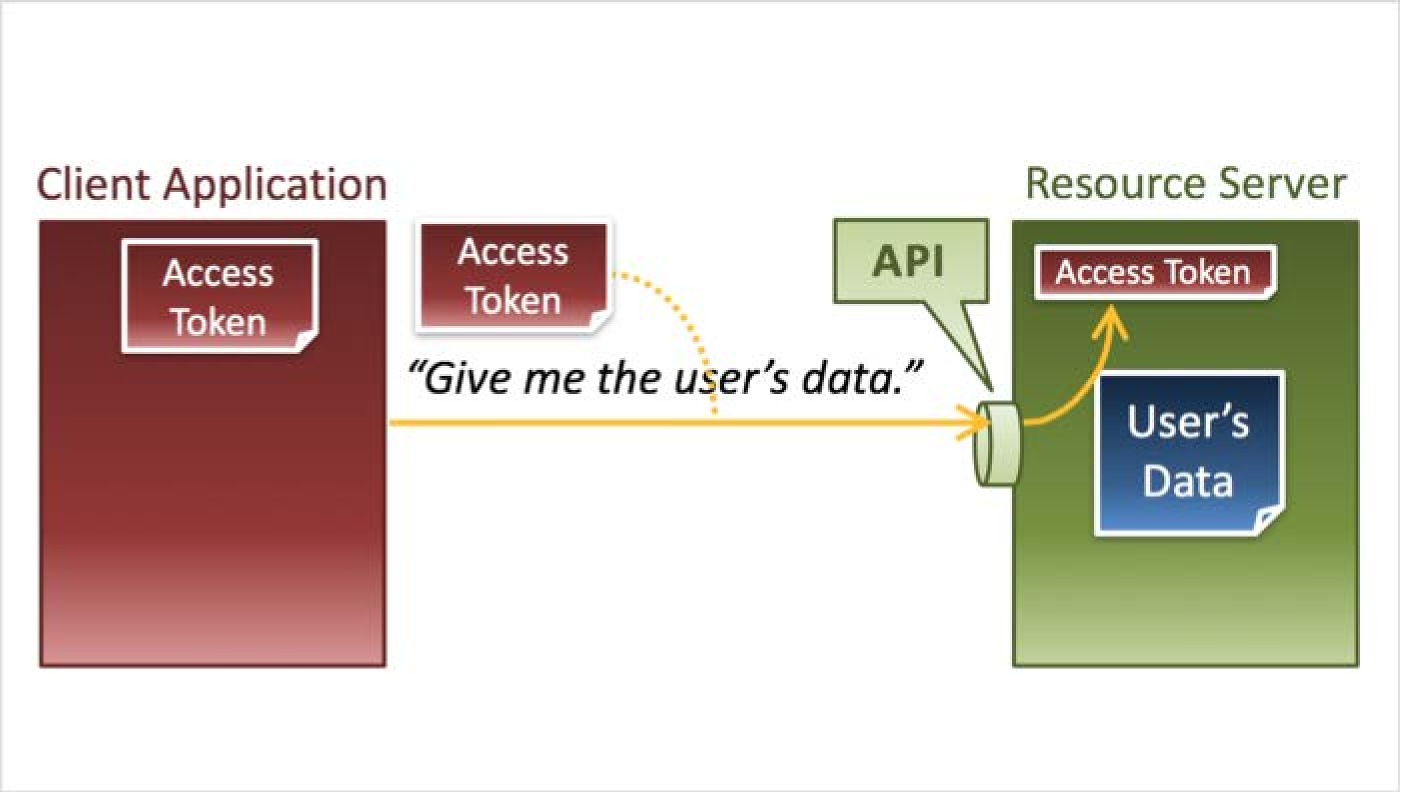
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

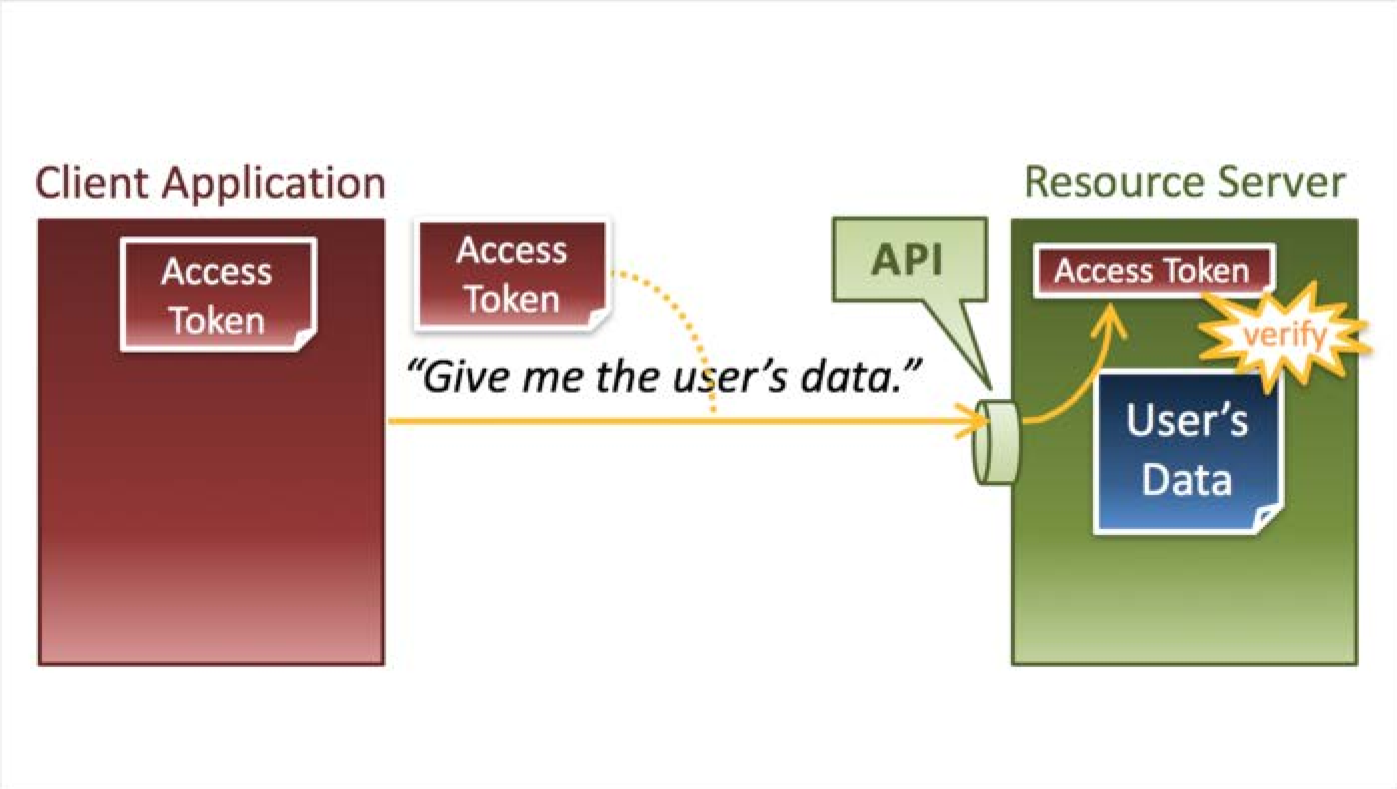
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

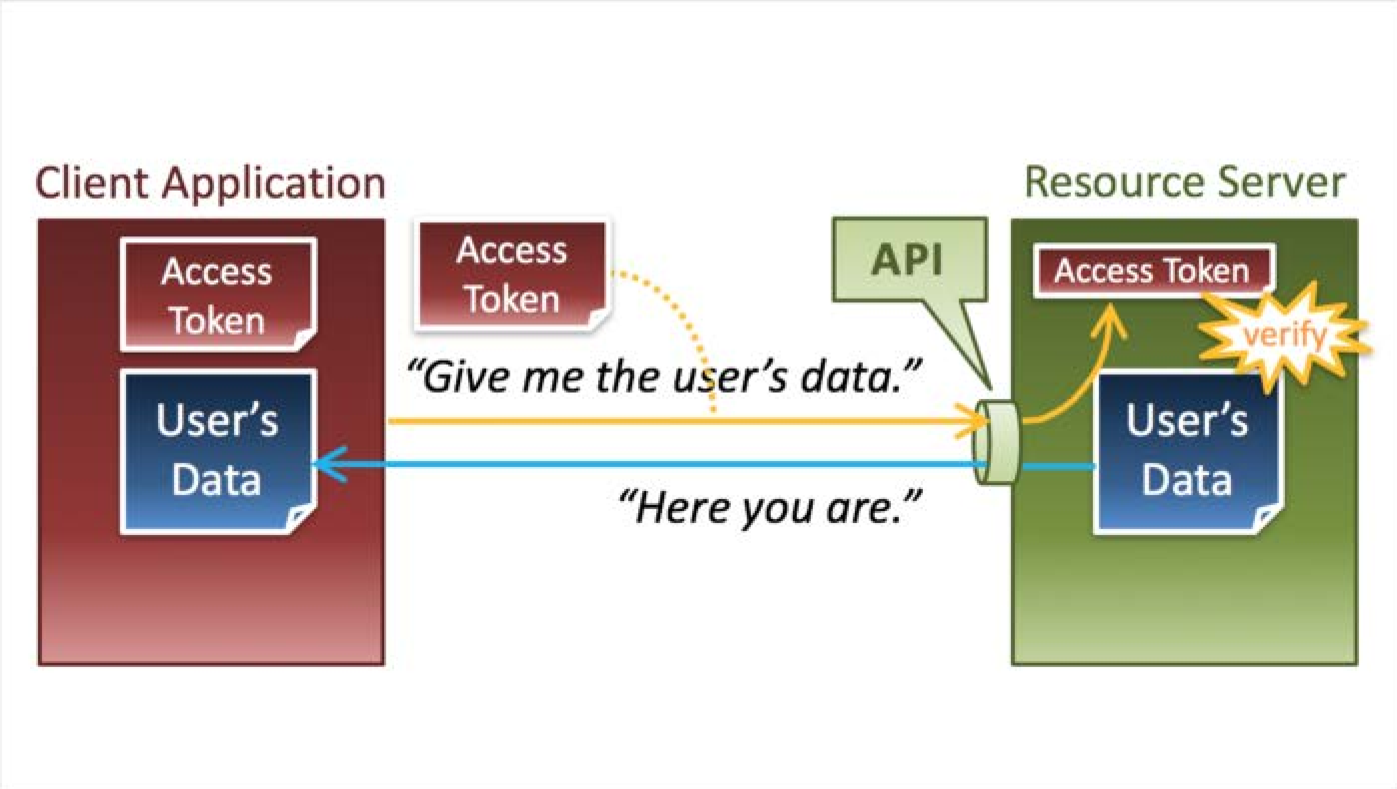
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

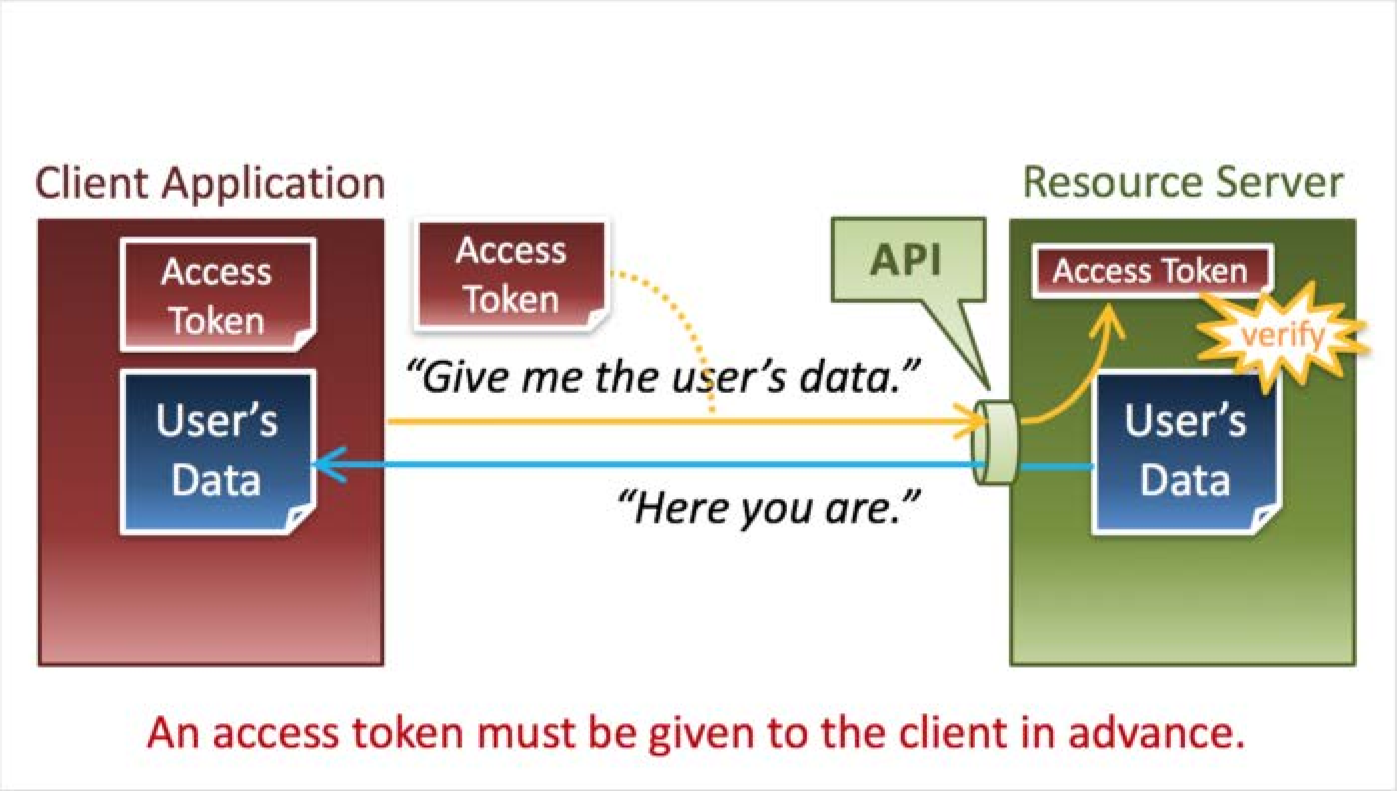
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).

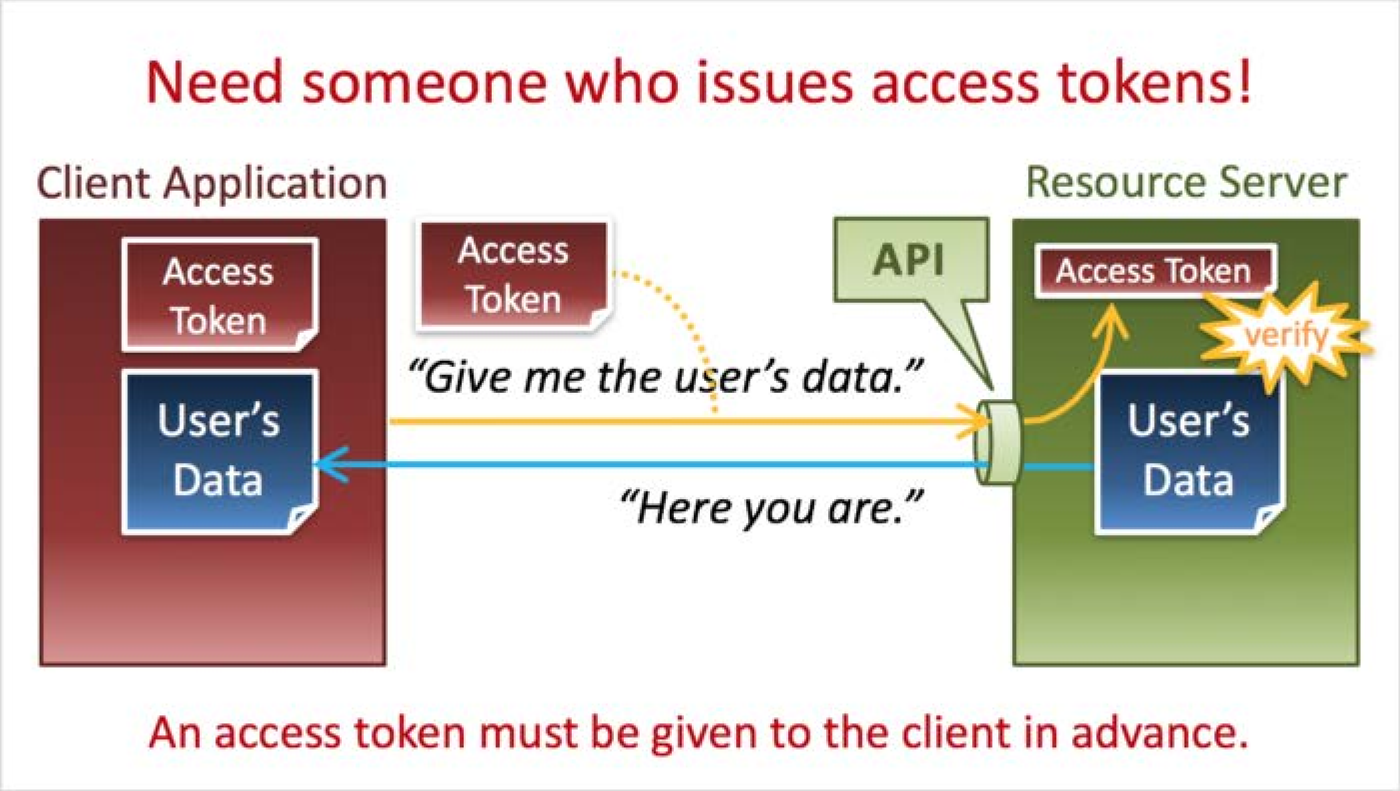


An access token must be given to the client in advance.

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

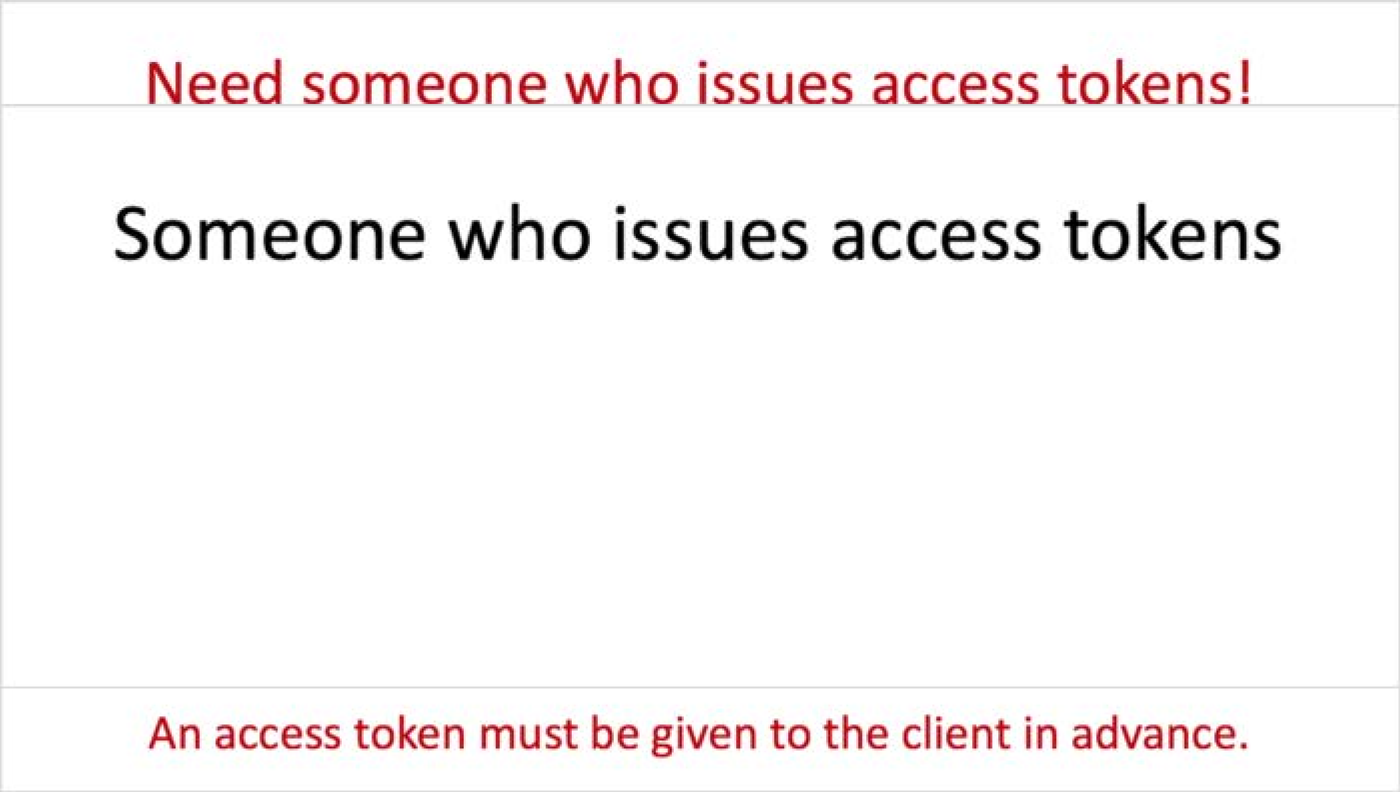
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).


Need someone who issues access tokens!
Client Application
Resource Server
Access Token
User's Data
Access Token
API
Access Token
verify
User's Data
"Give me the user's data."
"Here you are."
An access token must be given to the client in advance.

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



Need someone who issues access tokens!

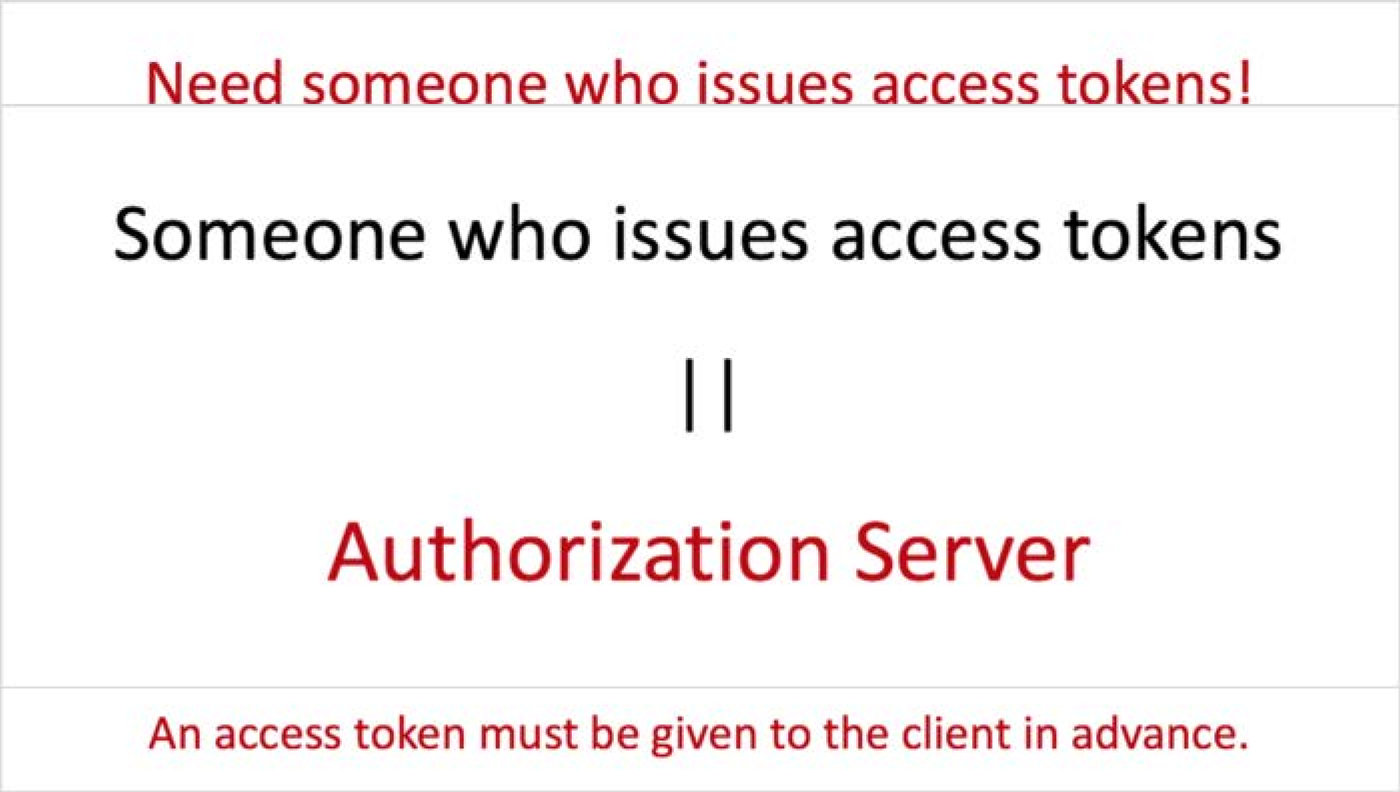
Someone who issues access tokens

An access token must be given to the client in advance.

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



Need someone who issues access tokens!

Someone who issues access tokens

||

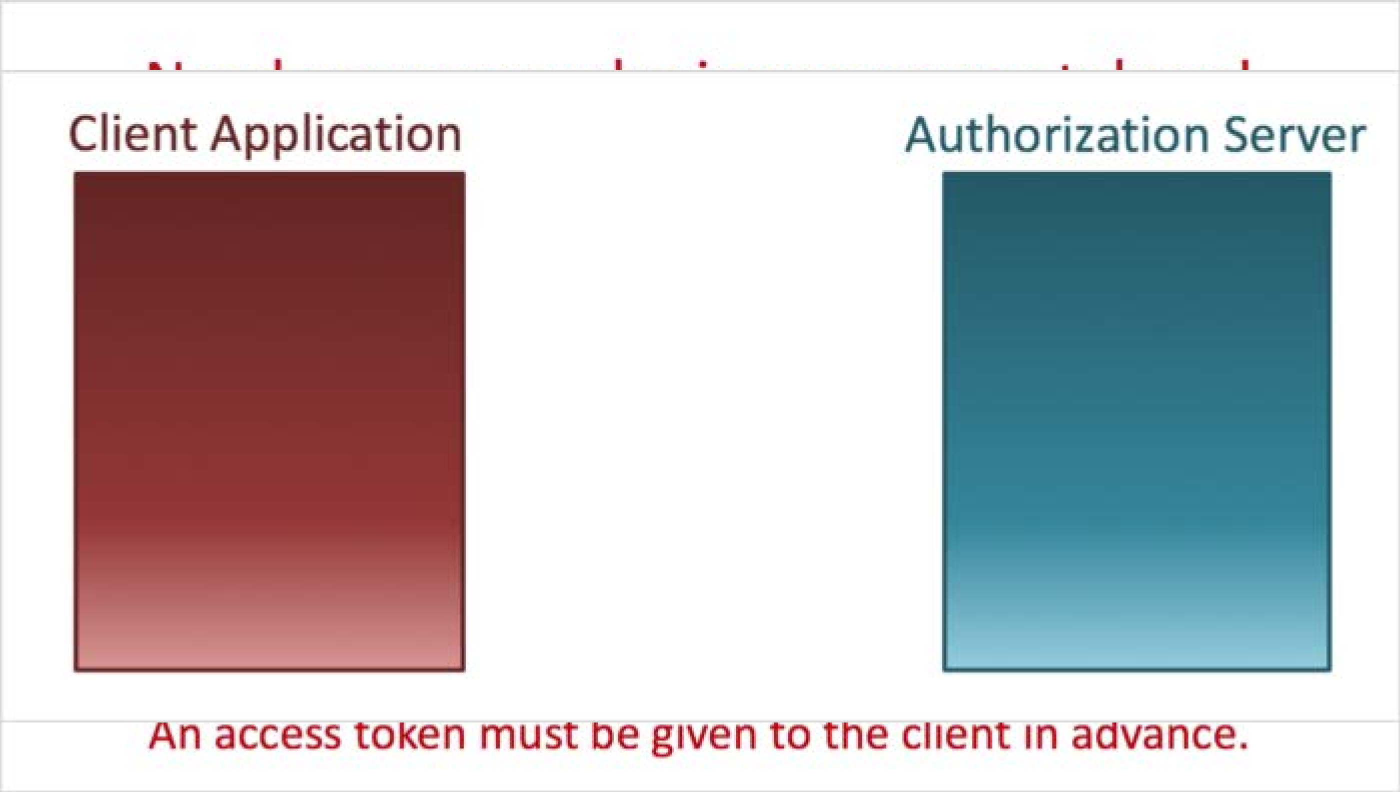
Authorization Server

An access token must be given to the client in advance.

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).

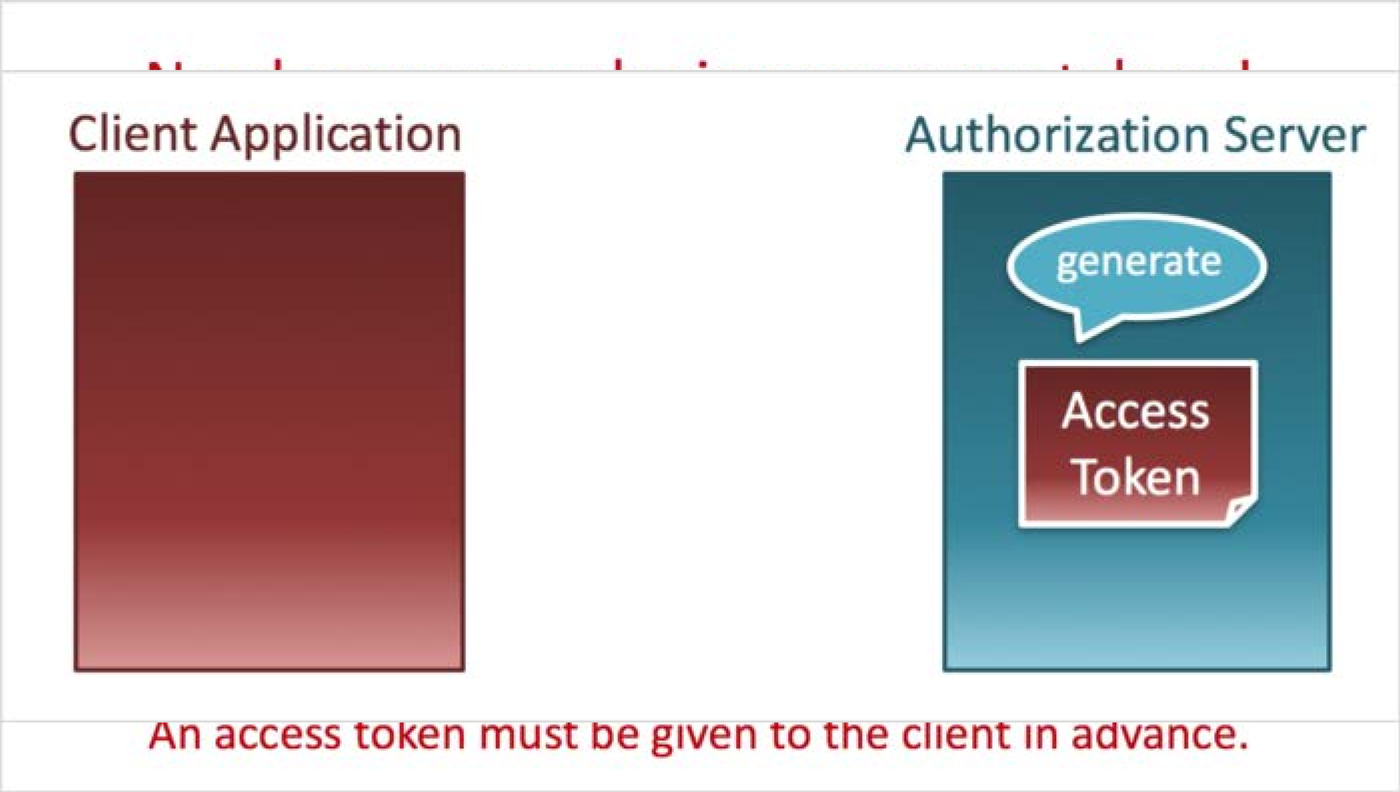


An access token must be given to the client in advance.

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).

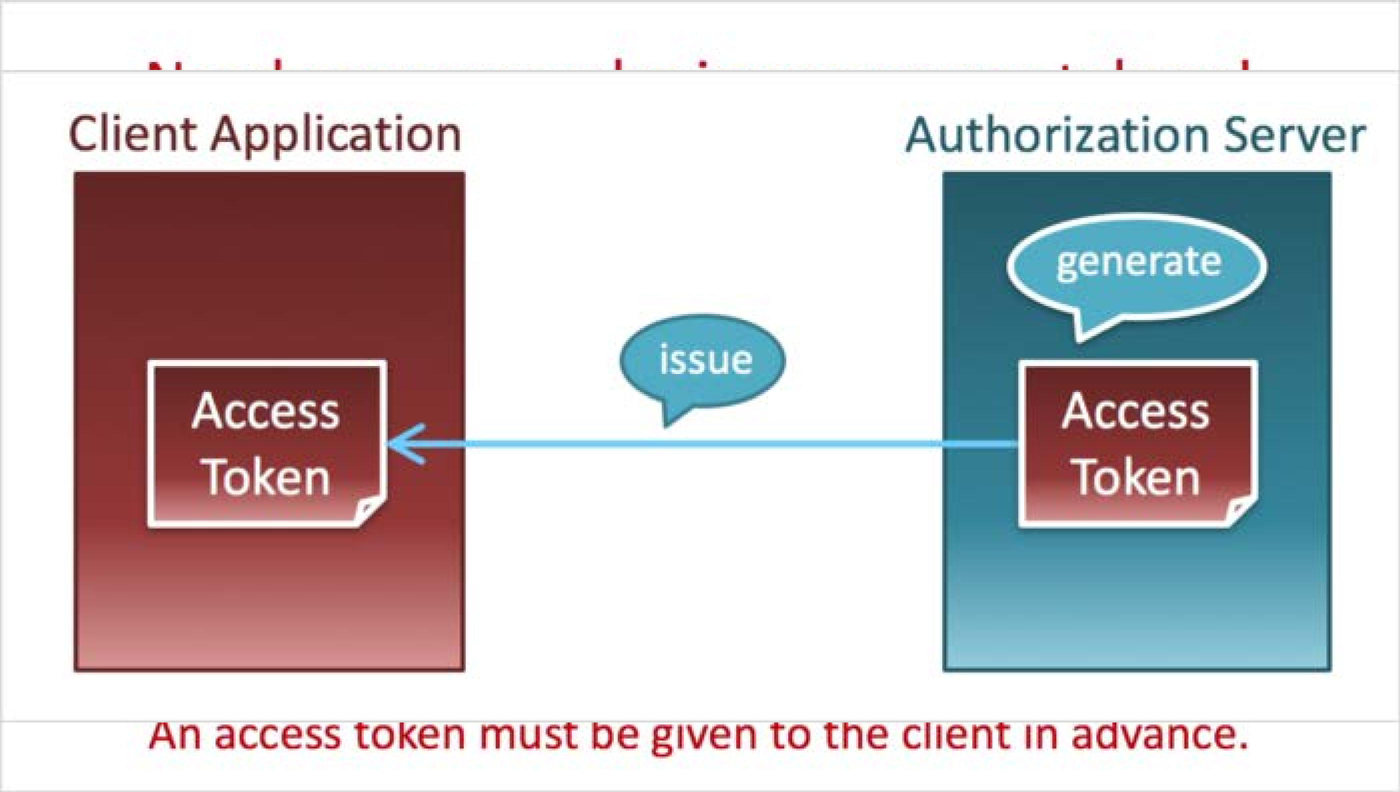


An access token must be given to the client in advance.

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

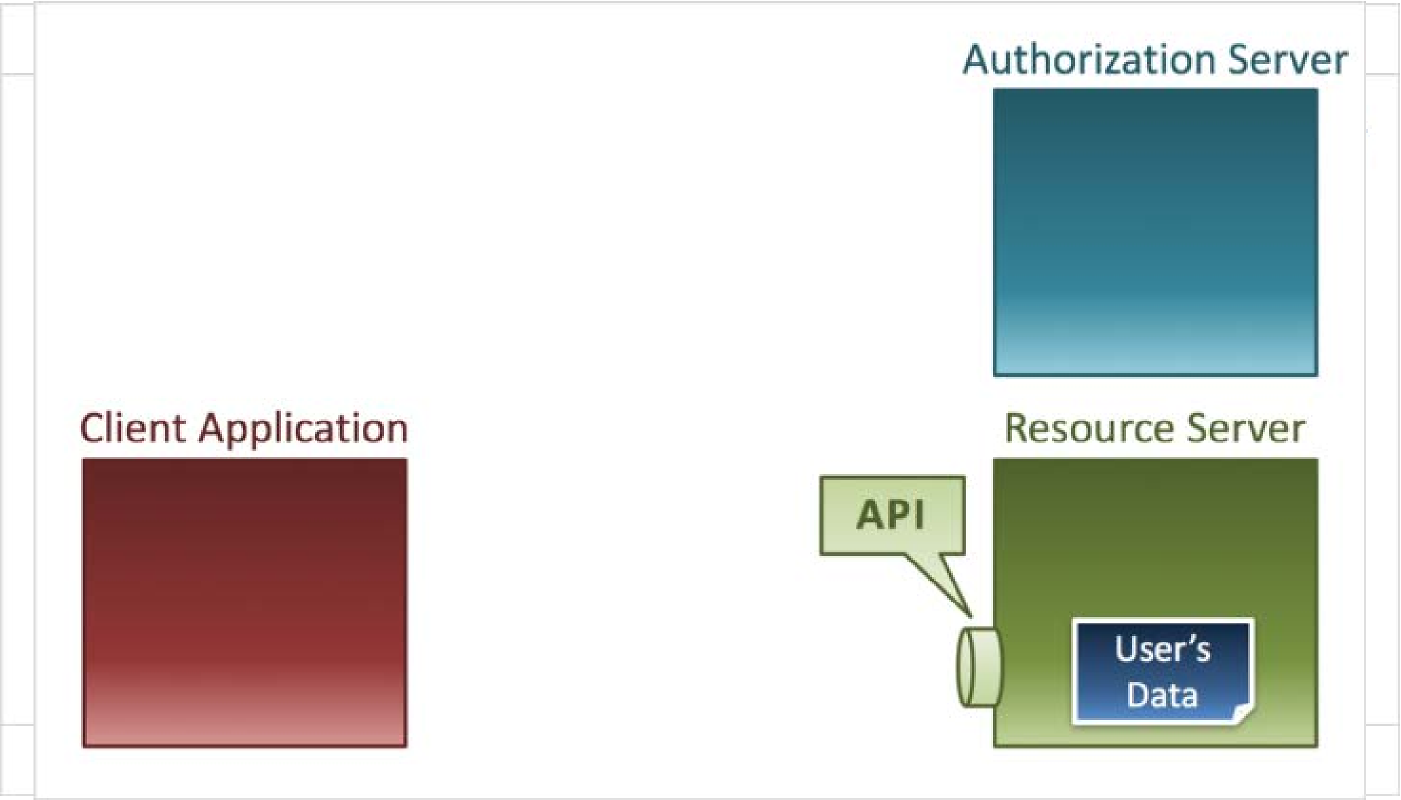
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

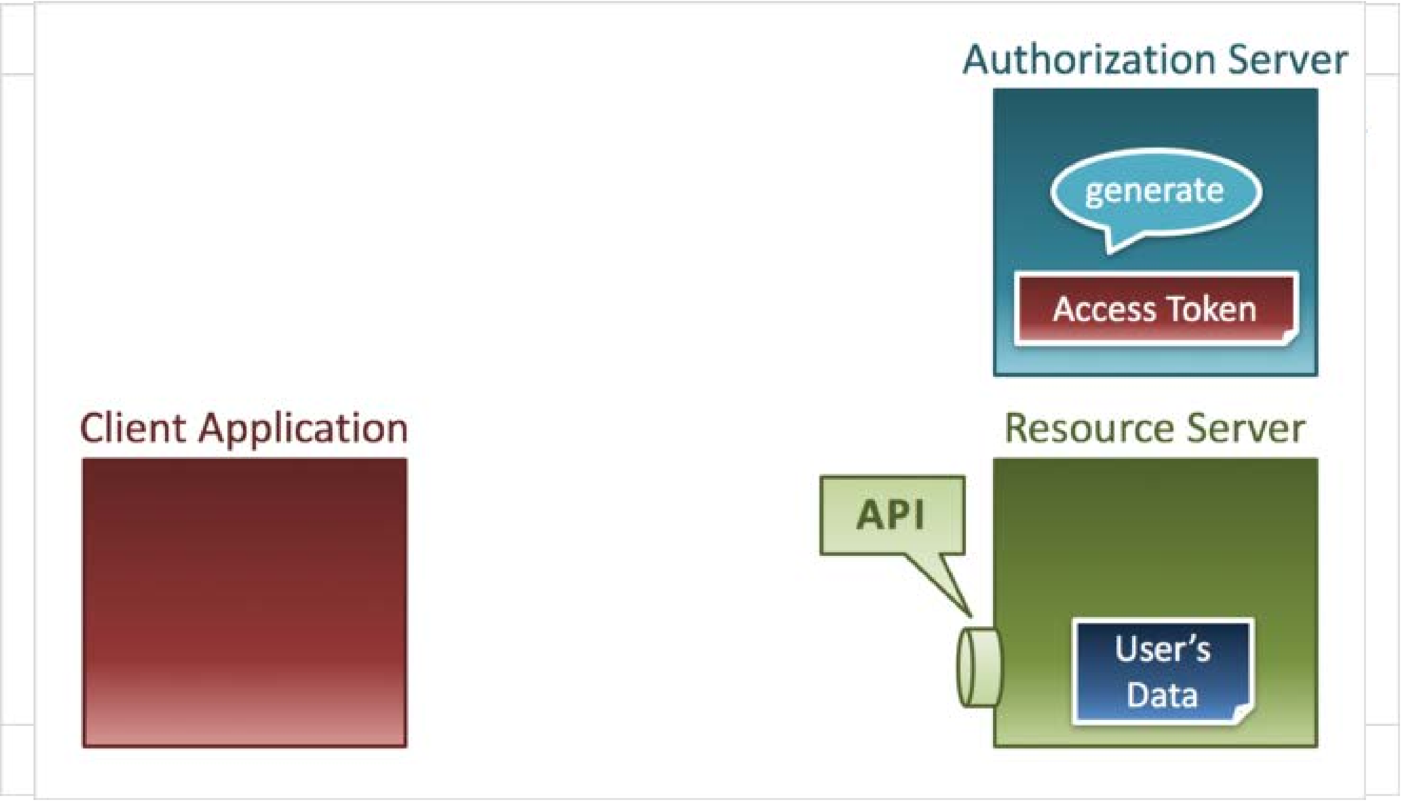
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

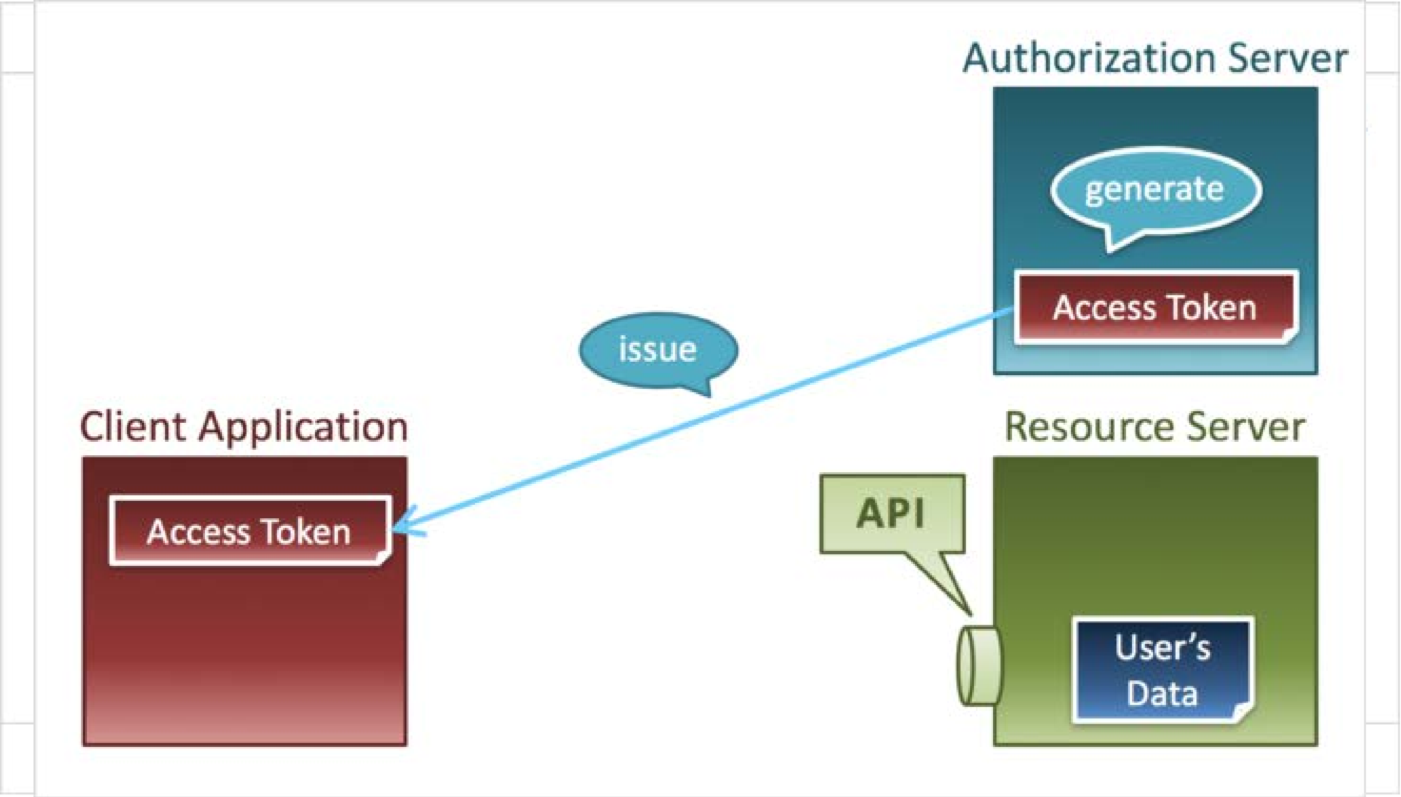
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

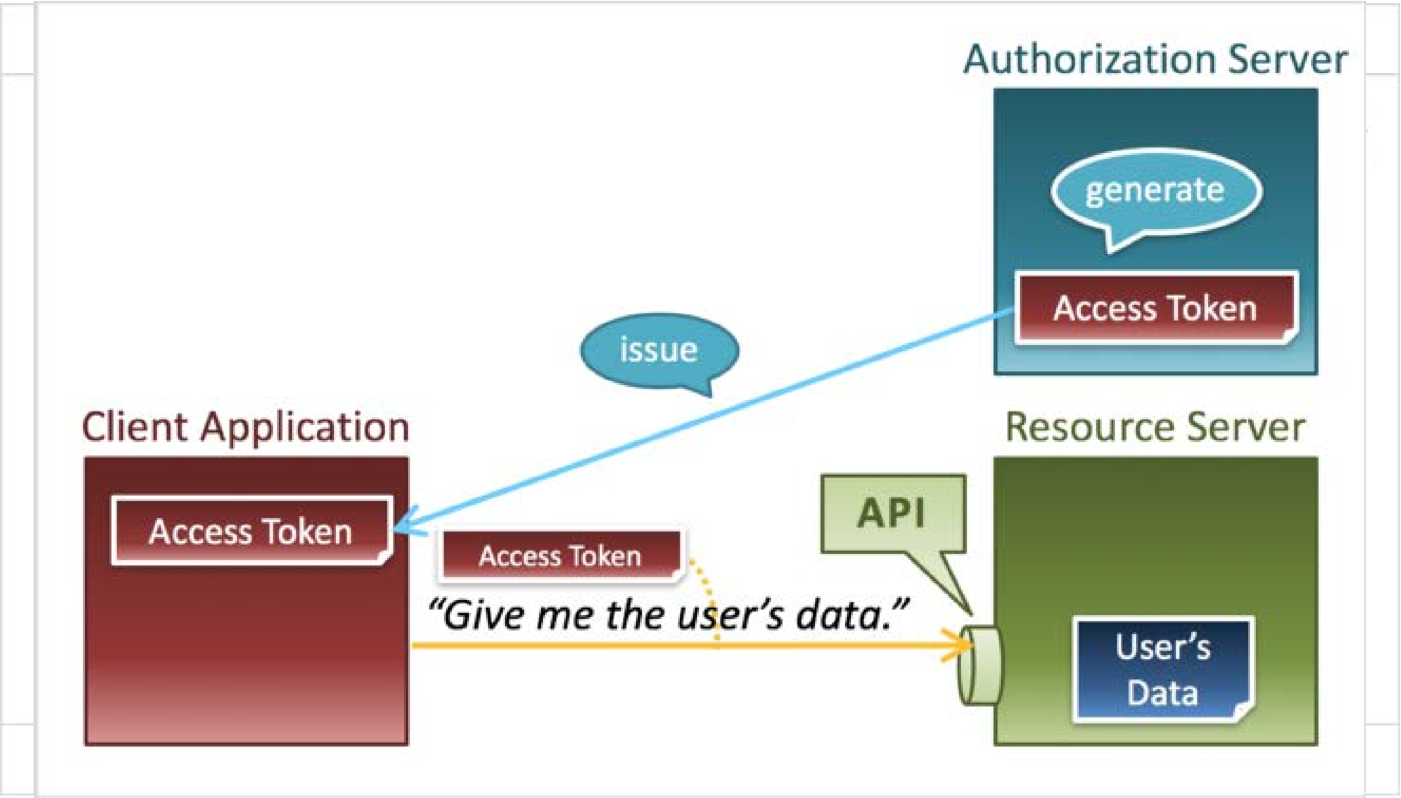
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

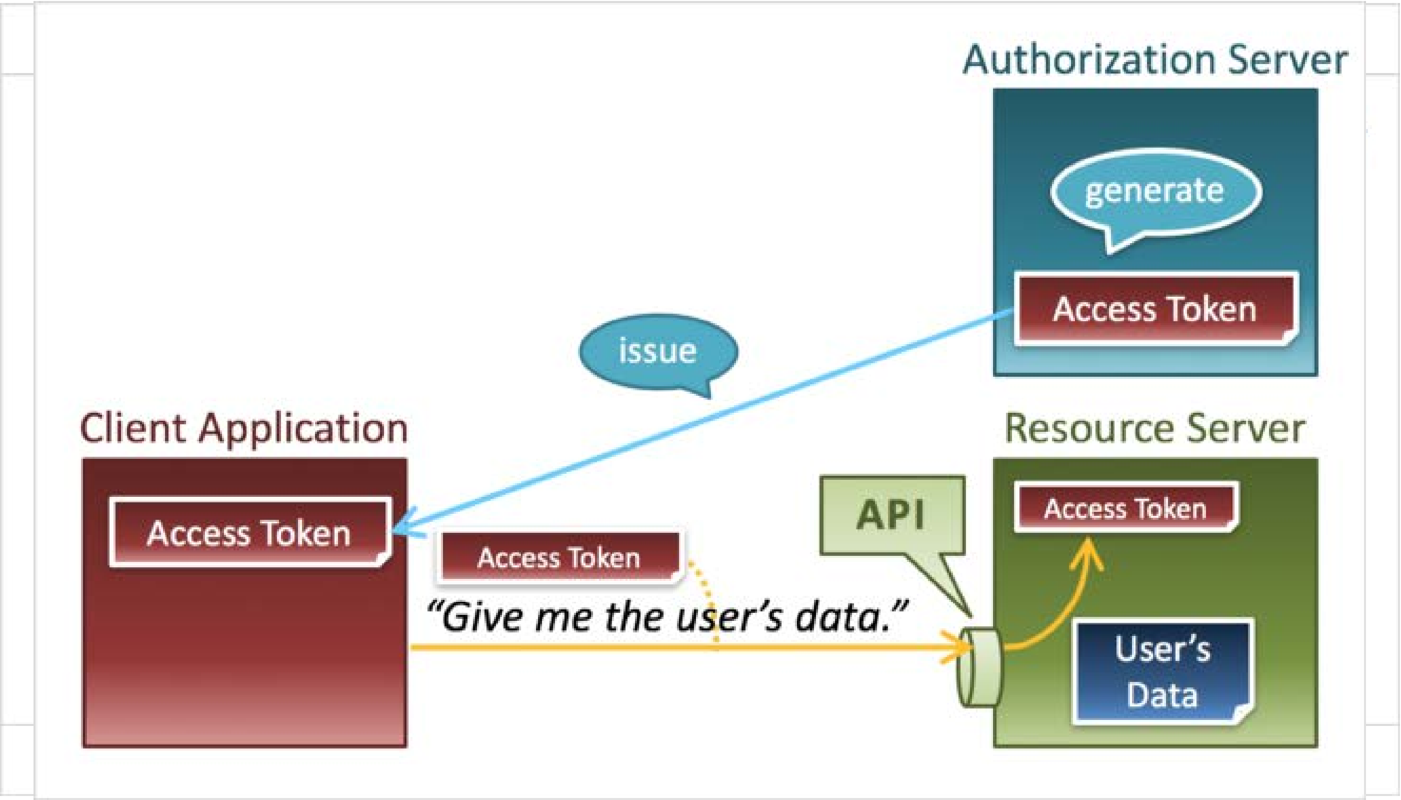
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

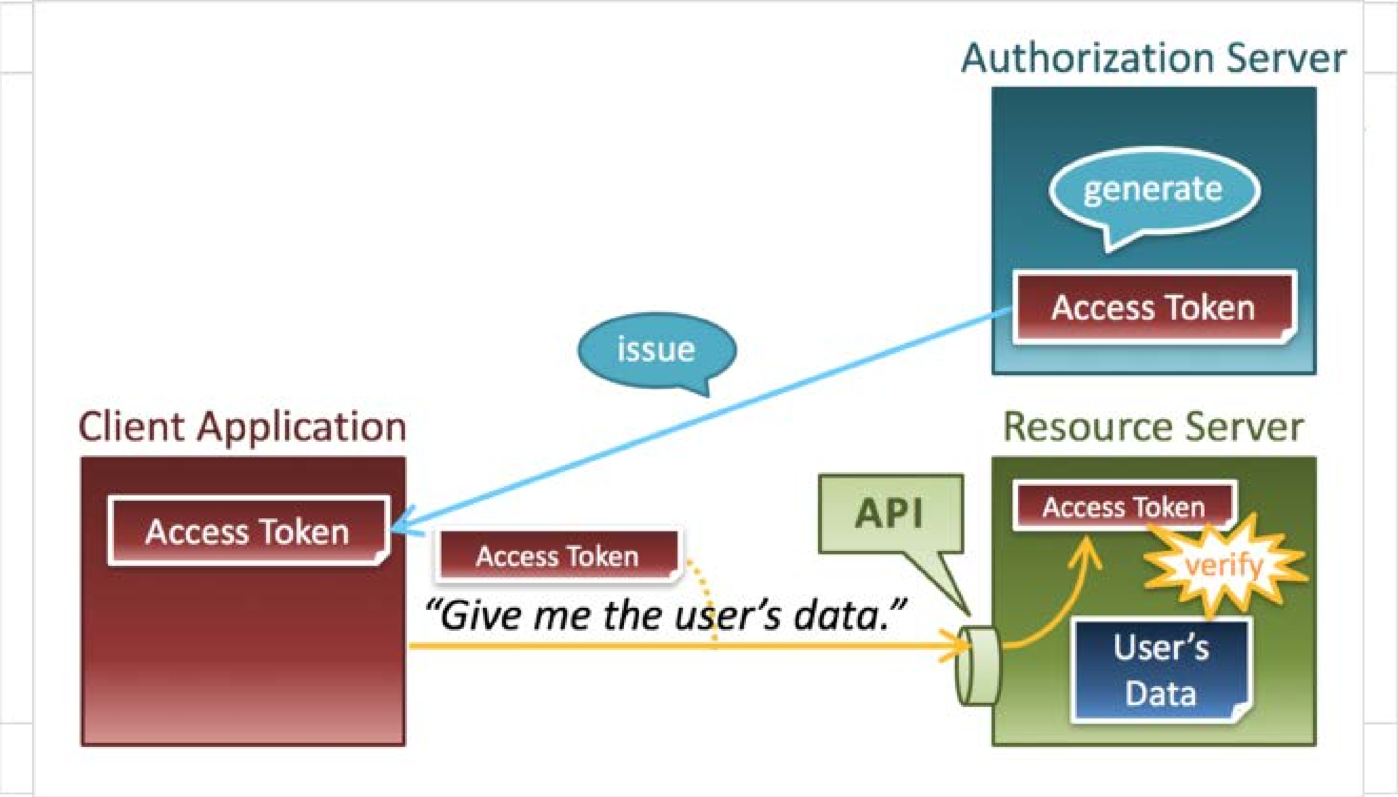
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

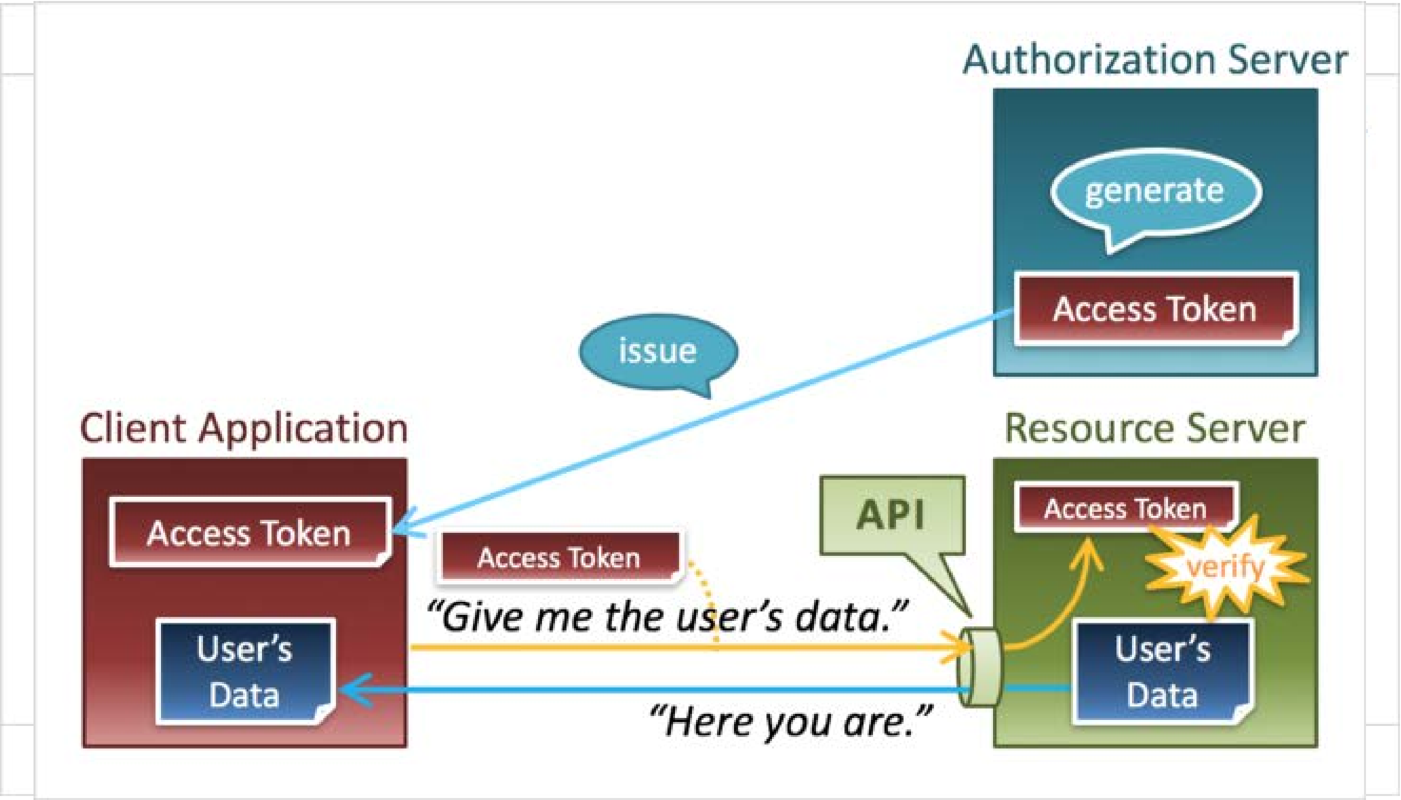
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

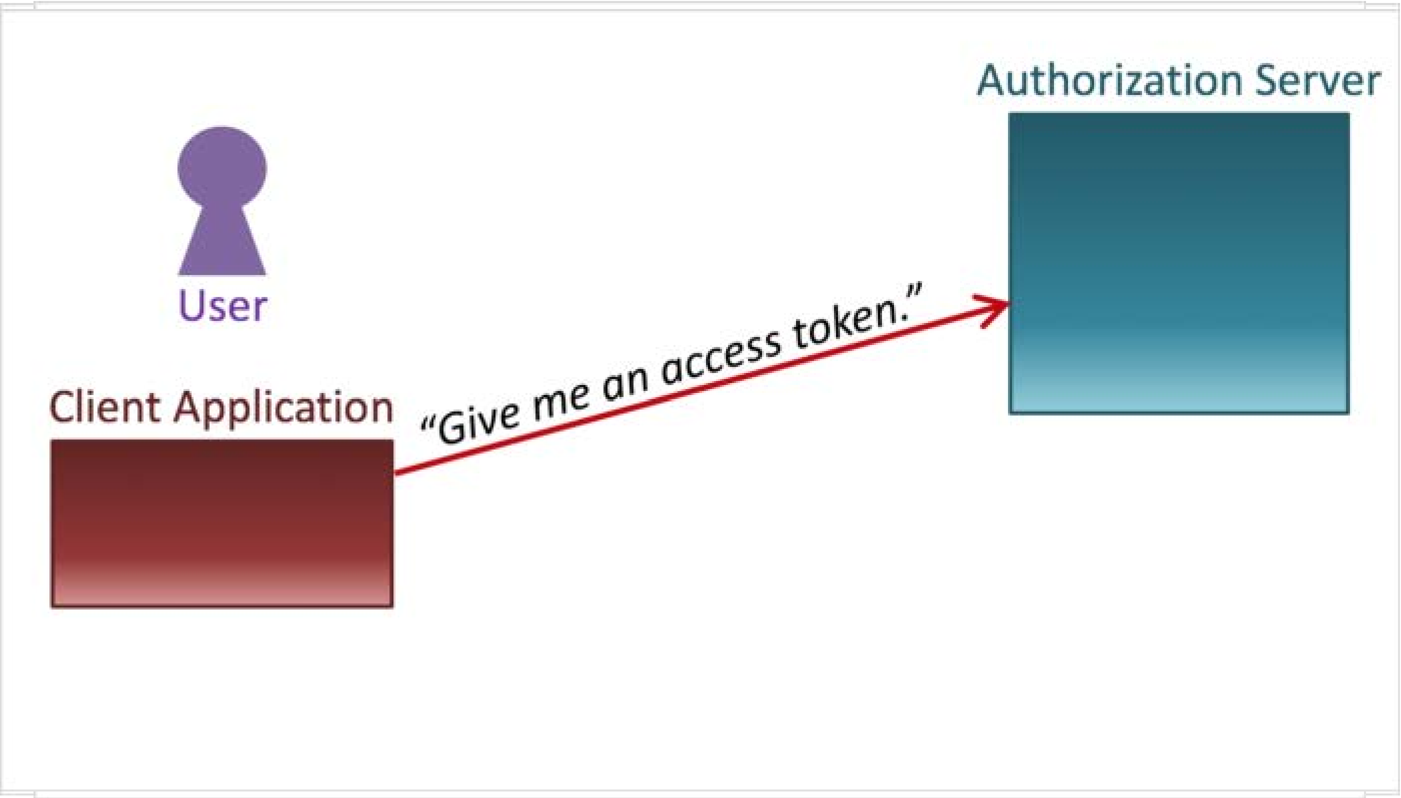
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).

Authorization Server
User
Client Application

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

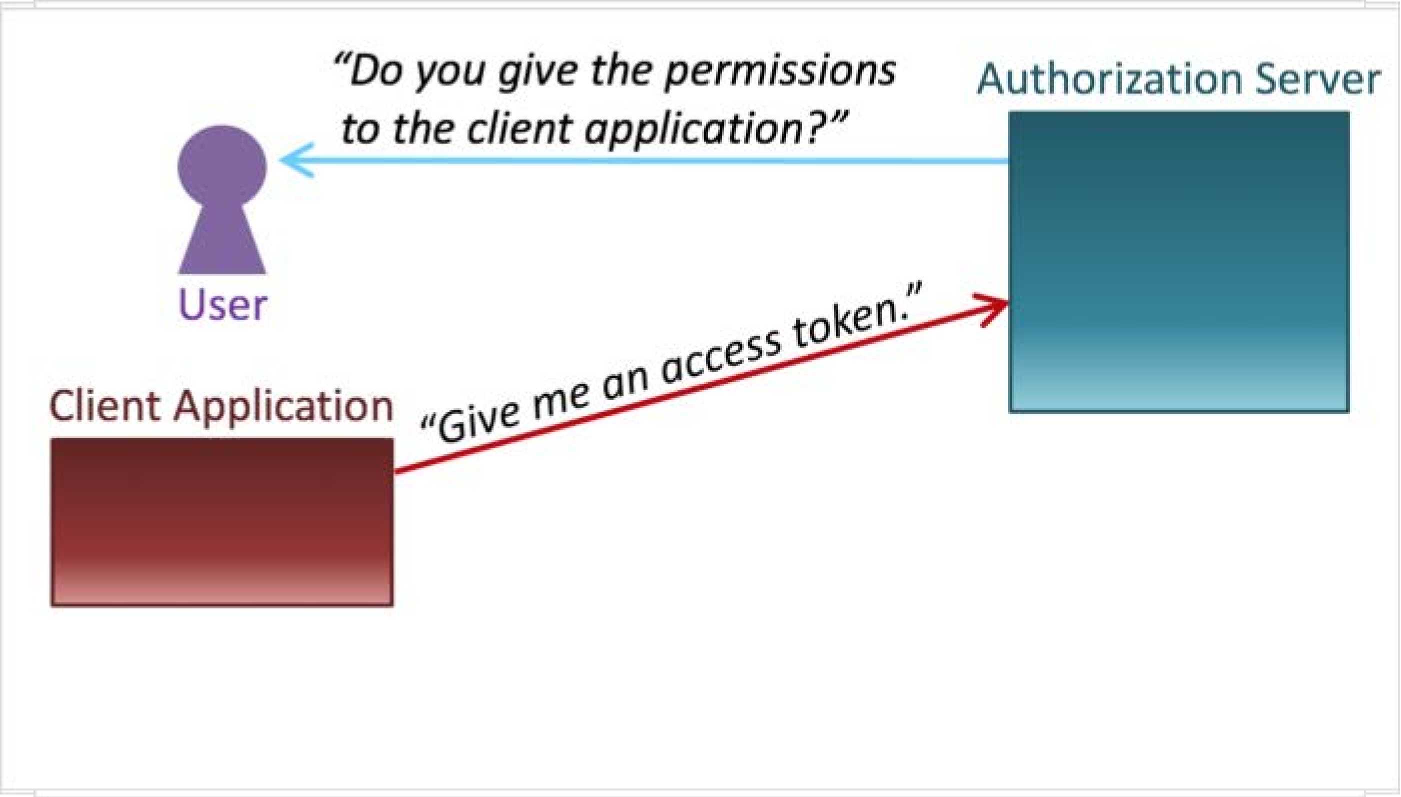
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

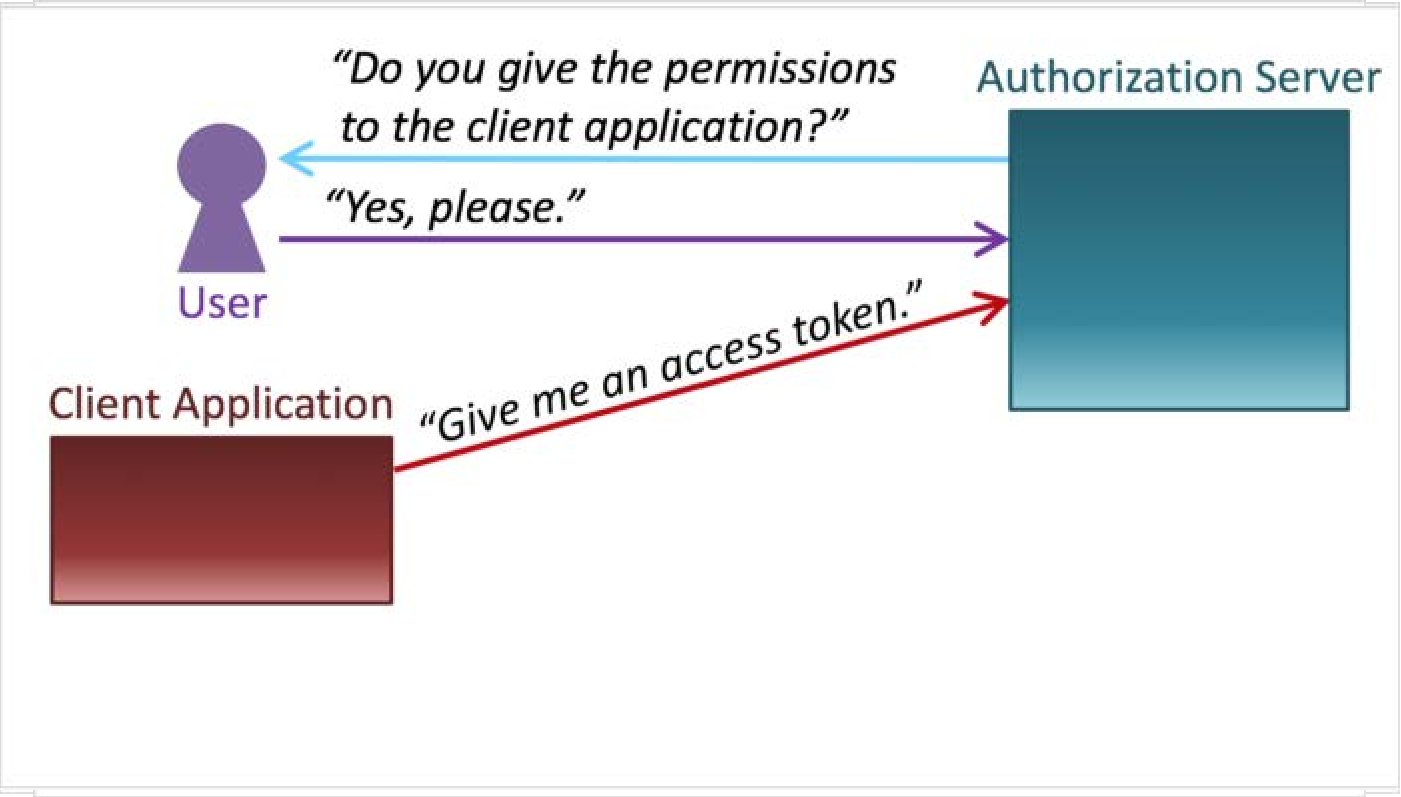
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).


"Do you give the permissions to the client application?"
Authorization Server
User
Client Application
"Give me an access token."

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

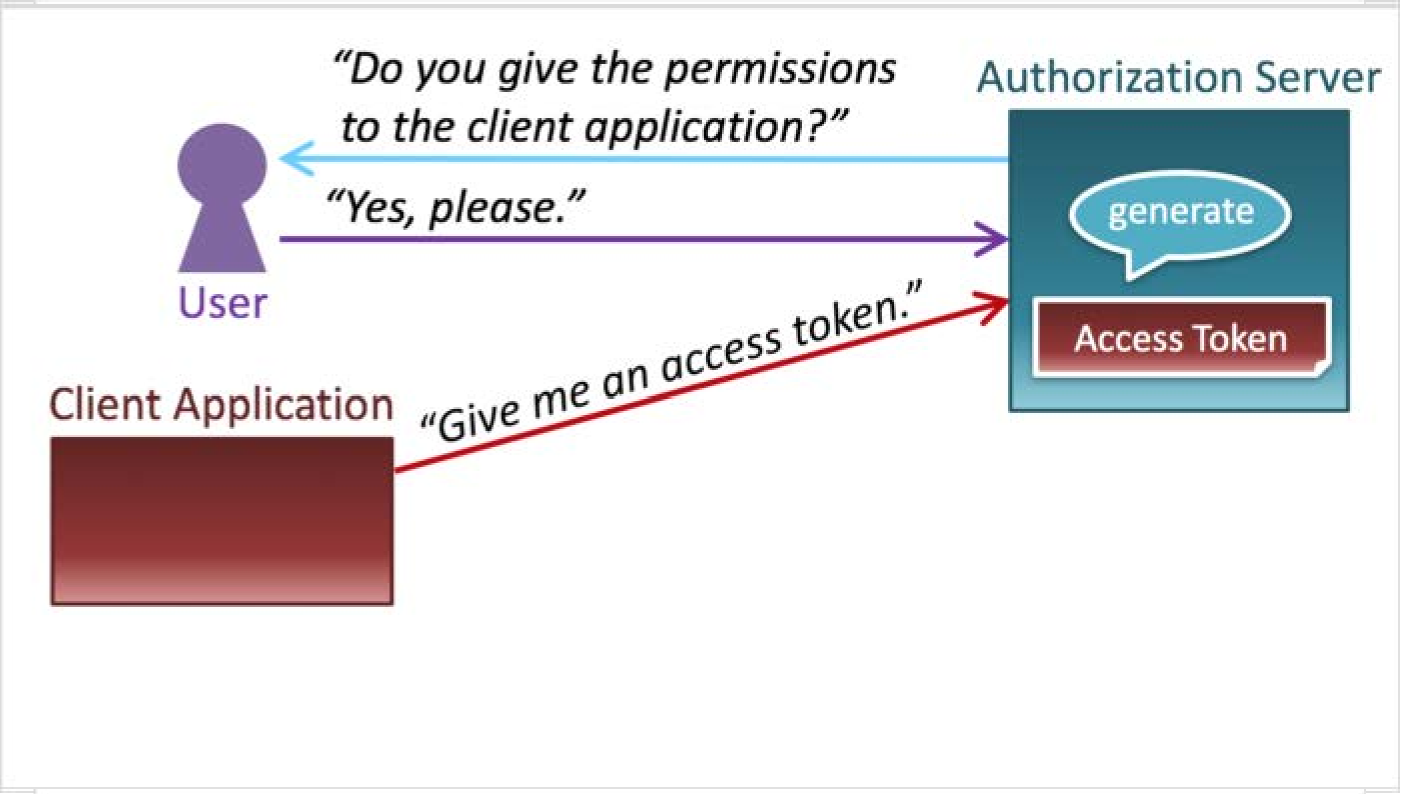
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).


"Do you give the permissions to the client application?"
Authorization Server
"Yes, please."
User
Client Application
"Give me an access token."

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

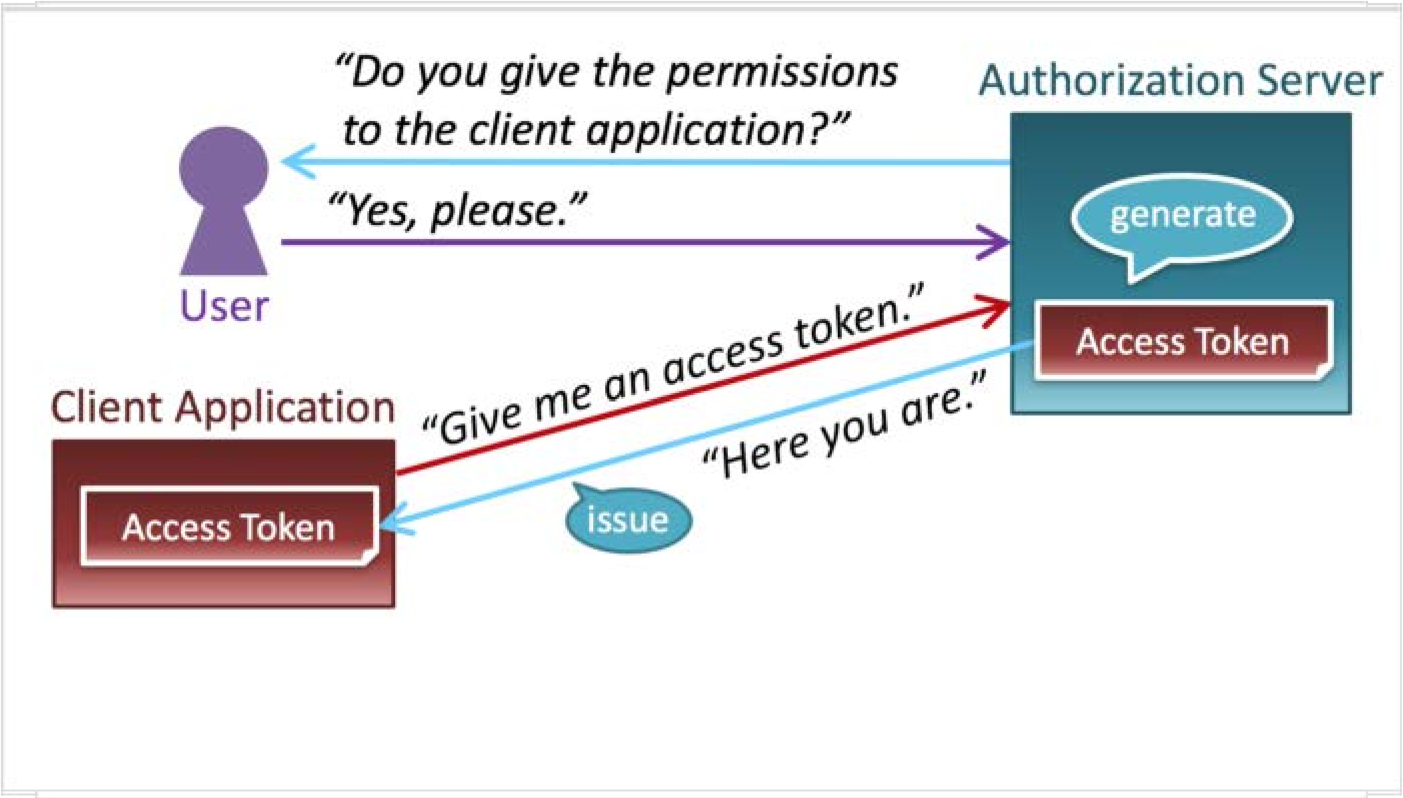
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).


"Do you give the permissions to the client application?"
Authorization Server
"Yes, please."
generate
User
Access Token
Client Application
"Give me an access token."

OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

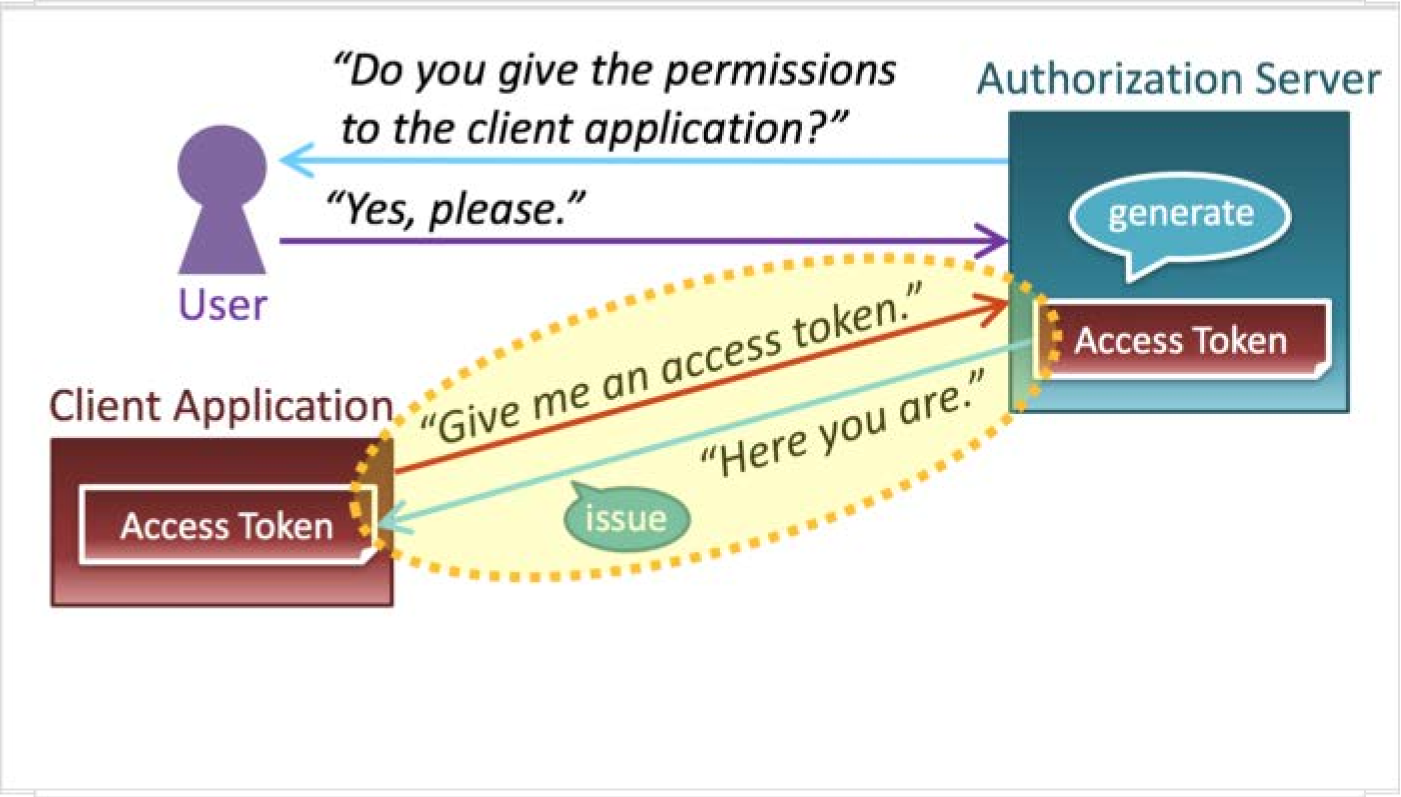
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

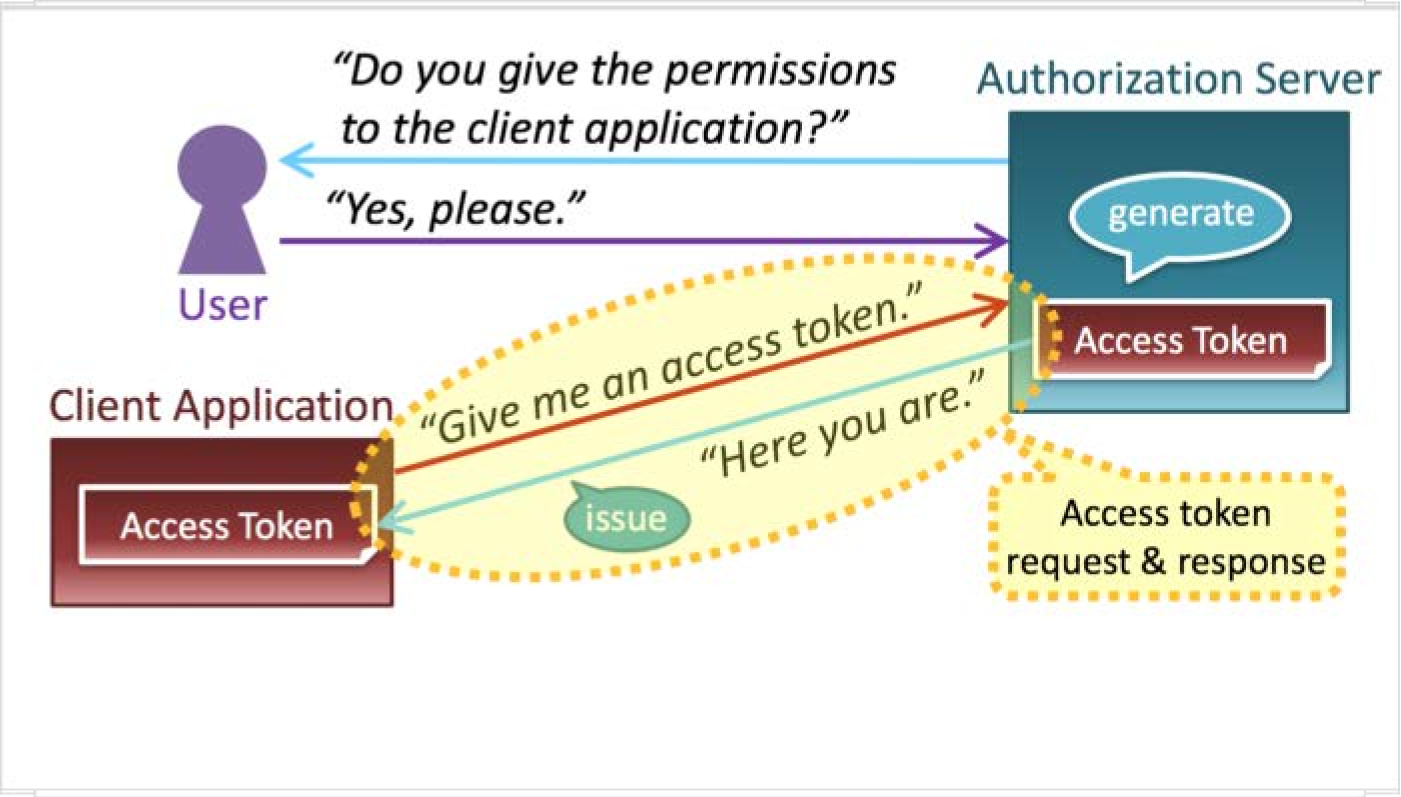
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).



OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

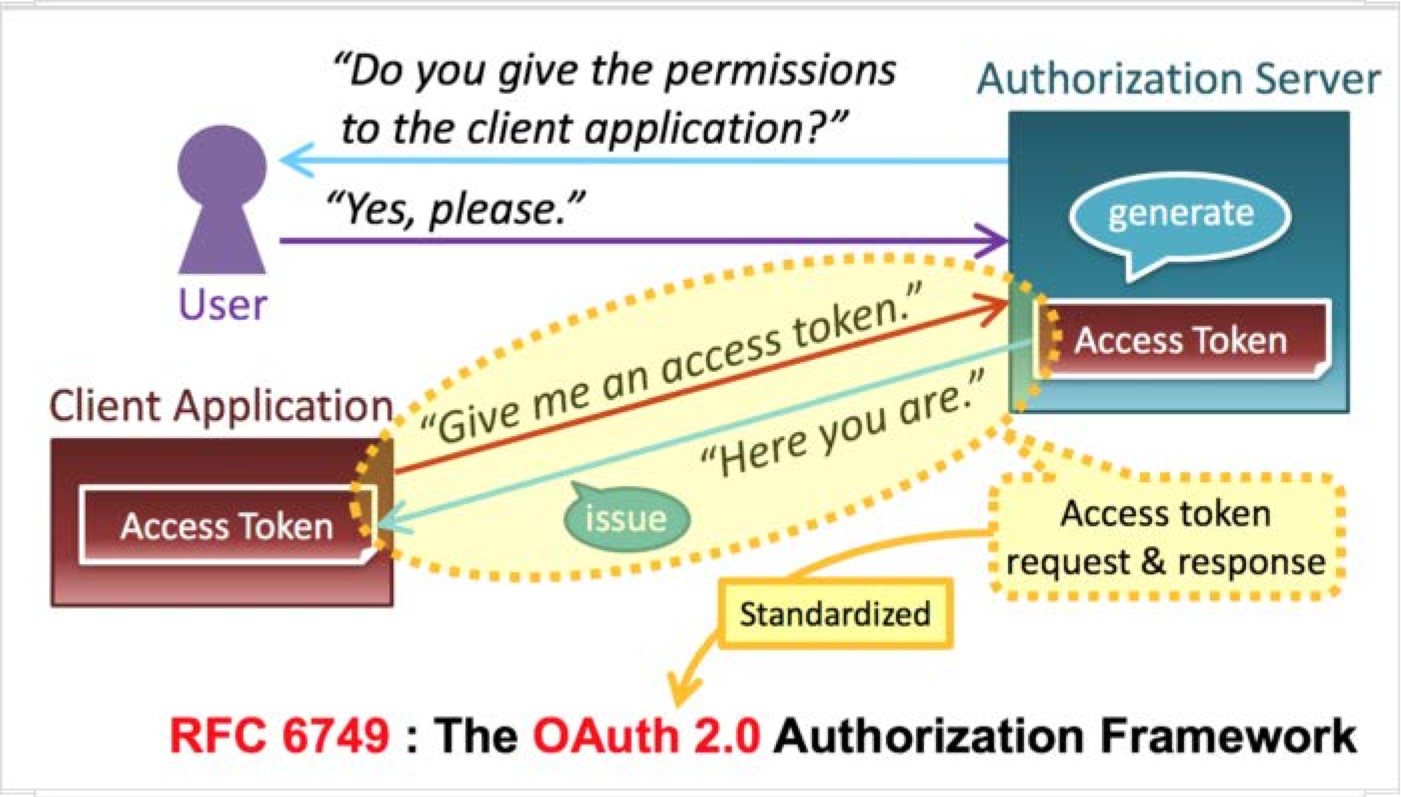
- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).


"Do you give the permissions to the client application?"
Authorization Server
User
"Yes, please."
generate
Access Token
Client Application
"Give me an access token."
"Here you are."
Access Token
issue
Access token request & response

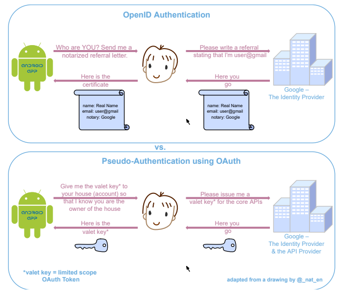
OAUTH 2.0

Bron: Takahiko Kawasaki (zie ook deze diagrammen)

- There are data of a user.
- There is a server which manages the user's data. The server is called "Resource Server".
- There is a "Client Application" which wants to use the user's data.
- Let's prepare a gate to pass the user's data through. The gate is called "API".
- The client application requests the user's data.
- The resource server returns the user's data.
- What if there is a malicious client application?
- Even if the client application that requests the user's data is a malicious one, ...
- ... the resource server returns the user's data.
- Even a malicious application can get the user's data.
- We need a mechanism to protect the user's data.
- In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.
- The client application presents the access token when it requests the user's data.
- The resource server extracts the access token that is included in the request, ...
- ... and confirms that the access token denotes that the client application has permissions to access the user's data.
- After the confirmation, the resource server returns the user's data.
- To make this mechanism work, an access token must be given to the client application in advance.
- Consequently, we need someone who issues access tokens.
- Someone who issues access tokens ...
- ... is called "Authorization Server".
- The relationship between a client application and an authorization server is as follows.
- An authorization server generates an access token ...
- ... and issues the access token to a client application.
- Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".
- The authorization server generates an access token ...
- ... and issues the access token to the client application.
- The client application requests the user's data with the access token.
- The resource server extracts the access token from the request, ...
- ... confirms that the access token has permissions to access the user's data ...
- ... and returns the user's data to the client application.
- In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.
- First, the client application requests an access token.
- Then, the authorization server asks the user whether to grant the requested permissions to the client application.
- If the user allows the authorization server to issue an access token to the client application, ...
- ... the authorization server generates an access token ...
- ... and issues the access token to the client application.
- By the way, pay attention to the part encircled by the yellow ellipse.
- The part represents an access token request and a response to the request.
- And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document, RFC 6749 (The OAuth 2.0 Authorization Framework).

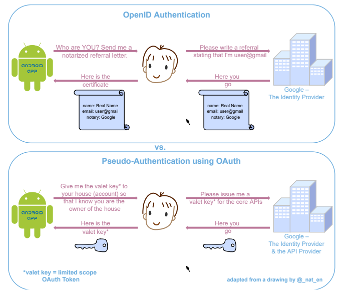

"Do you give the permissions to the client application?"
Authorization Server
User
"Yes, please."
generate
Access Token
Client Application
"Give me an access token."
"Here you are."
Access Token
issue
Access token request & response
Standardized
RFC 6749 : The OAuth 2.0 Authorization Framework

OAUTH EN OPENID


OpenID Authentication
android app
Who are YOU? Send me a notarized referral letter.
Here is the certificate
name: Real Name
email: user@gmail
notary: Google
Please write a referral stating that I'm user@gmail
Here you go
name: Real Name
email: user@gmail
notary: Google
Google – The Identity Provider
vs.
Pseudo-Authentication using OAuth
android app
Give me the valet key* to your house (account) so that I know you are the owner of the house
Here is the valet key*
Please issue me a valet key* for the core APIs
Here you go
Google – The Identity Provider & the API Provider
*valet key = limited scope OAuth Token
adapted from a drawing by @_nat_en

- De client logt dus niet in met Google gegevens in de webapplicatie, maar op Google zelf.
- De applicatie vertrouwt de “notaris” (in dit geval de Google Identity Provider)
- Een Identity provider is een Authorization server
- Tokens zijn vaak een beperkte tijd houdbaar
- Identity delegation
- Google/Facebook/LinkedIn/Twitter/…

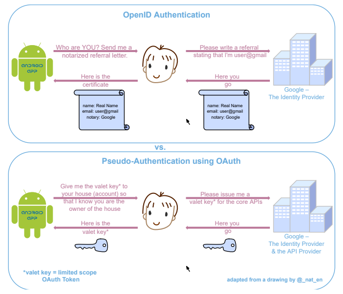
OAUTH EN OPENID



OpenID Authentication
Who are YOU? Send me a notarized referral letter.
Please write a referral stating that I'm user@gmail
android app
Here is the certificate
Here you go
Google – The Identity Provider
name: Real Name
email: user@gmail
notary: Google
name: Real Name
email: user@gmail
notary: Google
vs.
Pseudo-Authentication using OAuth
Give me the valet key* to your house (account) so that I know you are the owner of the house
Please issue me a valet key* for the core APIs
android app
Here is the valet key*
Here you go
Google – The Identity Provider & the API Provider
*valet key = limited scope OAuth Token
adapted from a drawing by @_nat_en

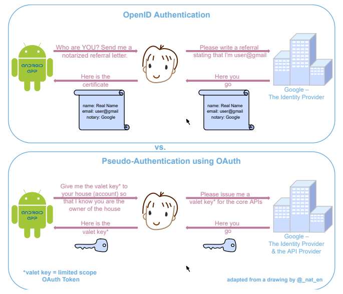
- De client logt dus niet in met Google gegevens in de webapplicatie, maar op Google zelf.
- De applicatie vertrouwt de “notaris” (in dit geval de Google Identity Provider)
- Een Identity provider is een Authorization server
- Tokens zijn vaak een beperkte tijd houdbaar
- Identity delegation
- Google/Facebook/LinkedIn/Twitter/...

OAUTH EN OPENID


OpenID Authentication
android app
Who are YOU? Send me a notarized referral letter.
Here is the certificate
name: Real Name
email: user@gmail
notary: Google
Please write a referral stating that I'm user@gmail
Here you go
name: Real Name
email: user@gmail
notary: Google
Google – The Identity Provider
vs.
Pseudo-Authentication using OAuth
android app
Give me the valet key* to your house (account) so that I know you are the owner of the house
Here is the valet key*
Please issue me a valet key* for the core APIs
Here you go
Google – The Identity Provider & the API Provider
*valet key = limited scope OAuth Token
adapted from a drawing by @_nat_en

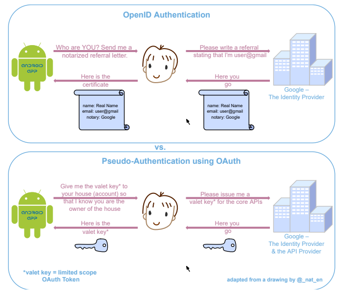
- De client logt dus niet in met Google gegevens in de webapplicatie, maar op Google zelf.
- De applicatie vertrouwt de “notaris” (in dit geval de Google Identity Provider)
- Een Identity provider is een Authorization server
- Tokens zijn vaak een beperkte tijd houdbaar
- Identity delegation
- Google/Facebook/LinkedIn/Twitter/…

OAUTH EN OPENID


OpenID Authentication
ANDROID APP
Who are YOU? Send me a notarized referral letter.
Here is the certificate
name: Real Name
email: user@gmail
notary: Google
Please write a referral stating that I'm user@gmail
Here you go
name: Real Name
email: user@gmail
notary: Google
Google – The Identity Provider
vs.
Pseudo-Authentication using OAuth
ANDROID APP
Give me the valet key* to your house (account) so that I know you are the owner of the house
Here is the valet key*
Please issue me a valet key* for the core APIs
Here you go
Google – The Identity Provider & the API Provider
*valet key = limited scope OAuth Token
adapted from a drawing by @_nat_en

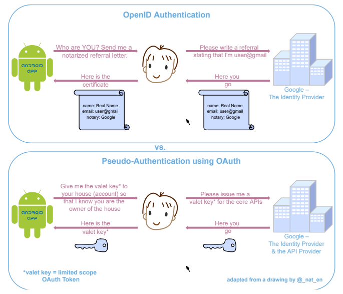
- De client logt dus niet in met Google gegevens in de webapplicatie, maar op Google zelf.
- De applicatie vertrouwt de “notaris” (in dit geval de Google Identity Provider)
- Een Identity provider is een Authorization server
- Tokens zijn vaak een beperkte tijd houdbaar
- Identity delegation
- Google/Facebook/LinkedIn/Twitter/…

OAUTH EN OPENID


OpenID Authentication
Who are YOU? Send me a notarized referral letter.
Please write a referral stating that I'm user@gmail
Here is the certificate
Here you go
name: Real Name
email: user@gmail
notary: Google
name: Real Name
email: user@gmail
notary: Google
ANDROID APP
Google – The Identity Provider
vs.
Pseudo-Authentication using OAuth
Give me the valet key* to your house (account) so that I know you are the owner of the house
Please issue me a valet key* for the core APIs
Here is the valet key*
Here you go
ANDROID APP
Google – The Identity Provider & the API Provider
*valet key = limited scope OAuth Token
adapted from a drawing by @_nat_en

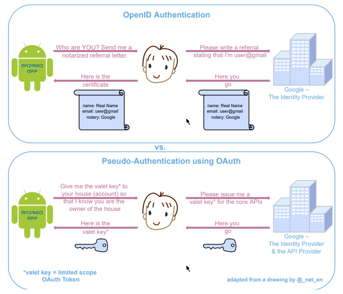
- De client logt dus niet in met Google gegevens in de webapplicatie, maar op Google zelf.
- De applicatie vertrouwt de “notaris” (in dit geval de Google Identity Provider)
- Een Identity provider is een Authorization server
- Tokens zijn vaak een beperkte tijd houdbaar
- Identity delegation
- Google/Facebook/LinkedIn/Twitter/…

OAUTH EN OPENID


OpenID Authentication
android app
Who are YOU? Send me a notarized referral letter.
Please write a referral stating that I'm user@gmail
Here is the certificate
Here you go
name: Real Name
email: user@gmail
notary: Google
name: Real Name
email: user@gmail
notary: Google
Google – The Identity Provider
vs.
Pseudo-Authentication using OAuth
android app
Give me the valet key* to your house (account) so that I know you are the owner of the house
Please issue me a valet key* for the core APIs
Here is the valet key*
Here you go
Google – The Identity Provider & the API Provider
*valet key = limited scope OAuth Token
adapted from a drawing by @_nat_en

- De client logt dus niet in met Google gegevens in de webapplicatie, maar op Google zelf.
- De applicatie vertrouwt de “notaris” (in dit geval de Google Identity Provider)
- Een Identity provider is een Authorization server
- Tokens zijn vaak een beperkte tijd houdbaar
- Identity delegation
- Google/Facebook/LinkedIn/Twitter/…

OAUTH EN OPENID


OpenID Authentication
Who are YOU? Send me a notarized referral letter.
Here is the certificate
name: Real Name
email: user@gmail
notary: Google
Please write a referral stating that I'm user@gmail
Here you go
name: Real Name
email: user@gmail
notary: Google
ANDROID APP
Google –
The Identity Provider
vs.
Pseudo-Authentication using OAuth
Give me the valet key* to your house (account) so that I know you are the owner of the house
Here is the valet key*
Please issue me a valet key* for the core APIs
Here you go
ANDROID APP
Google –
The Identity Provider
& the API Provider
*valet key = limited scope OAuth Token
adapted from a drawing by @_nat_en

- De client logt dus niet in met Google gegevens in de webapplicatie, maar op Google zelf.
- De applicatie vertrouwt de “notaris” (in dit geval de Google Identity Provider)
- Een Identity provider is een Authorization server
- Tokens zijn vaak een beperkte tijd houdbaar
- Identity delegation
- Google/Facebook/LinkedIn/Twitter/...

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

-
  - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

-
  - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

-
  - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

-
  - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

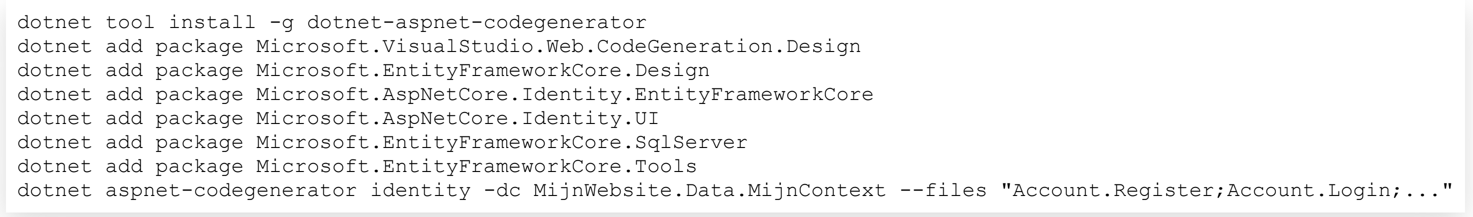
- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

-
  - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:



```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

-
  - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

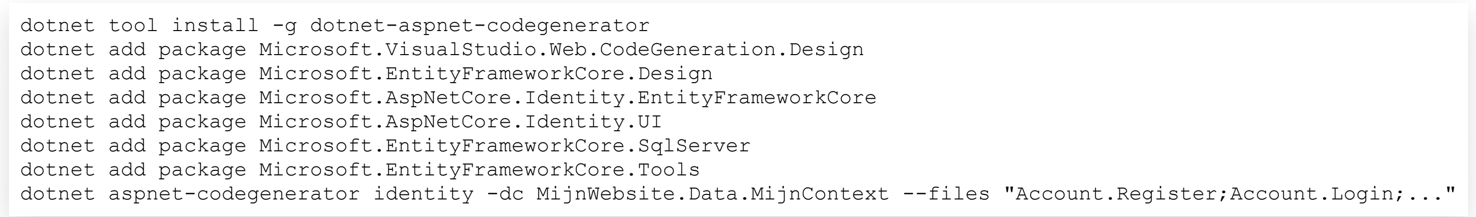
- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

- ▪ Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

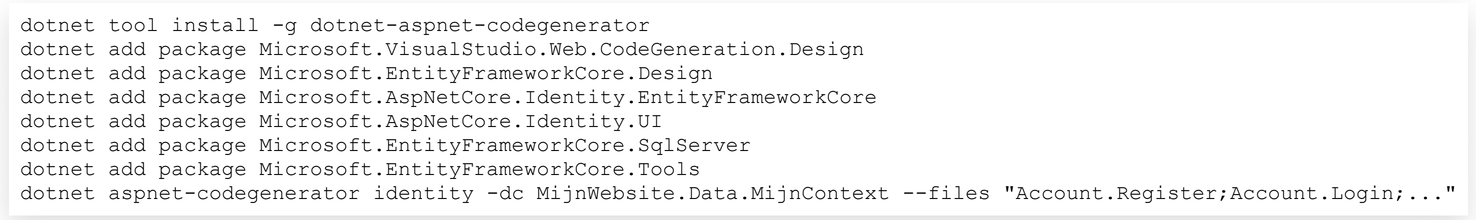


```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

-
  - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

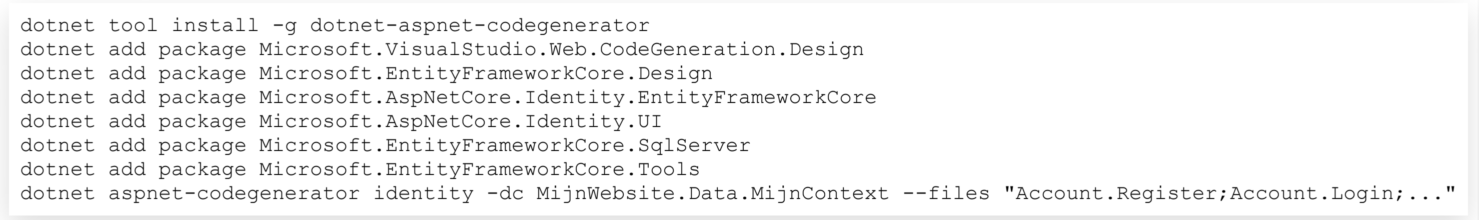


```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

-
  - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

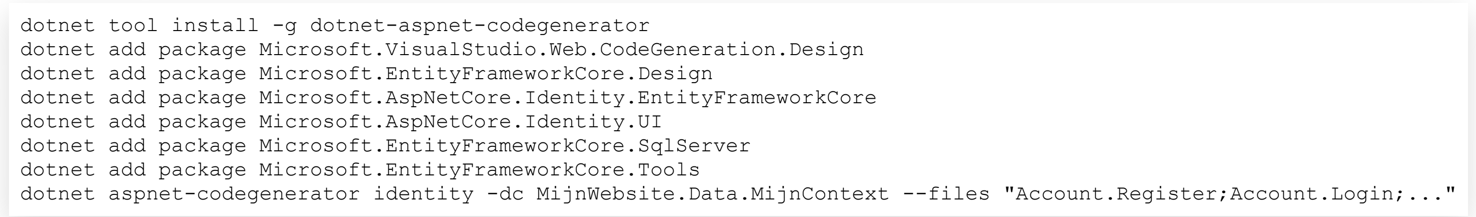


```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

-
  - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:



```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

- - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

- - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

- - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

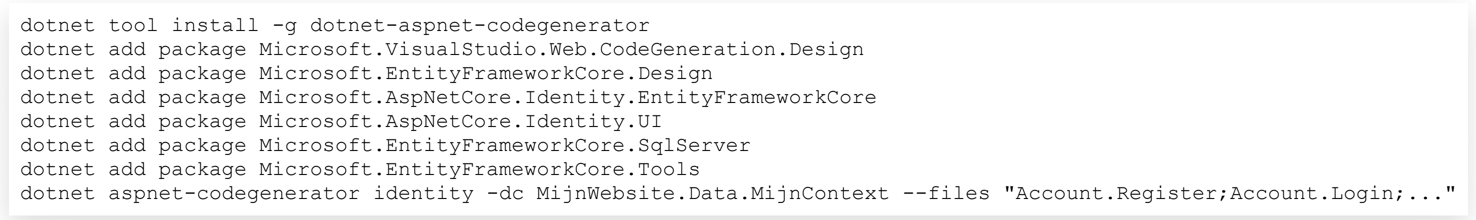
- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

- - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:



```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

- - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

≡

AUTHENTICATIE IN ASP.NET CORE (LITERATUUR)

- **ASP.NET Core Identity** is een kant-en-klare manier om inloggen/registreren te regelen:
  - Frontend in **Razor Pages**, en
  - Backend in EF Core
- In Visual Studio: demo. In VS Code:

```
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
dotnet add package Microsoft.AspNetCore.Identity.UI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet aspnet-codegenerator identity -dc MijnWebsite.Data.MijnContext --files "Account.Register;Account.Login;..."
```

-
  - Voer de migratie uit en update de database
  - Voeg `services.AddRazorPages();` toe aan `Startup.ConfigureServices`
  - Voeg `app.UseAuthentication();` toe aan `Startup.Configure`
  - Voeg `endpoints.MapRazorPages();` toe bij de endpoints gedefinieerd in `Startup.Configure`
- Zelf maken is erg ingewikkeld!
  - salted gehasht opslaan van de wachtwoorden.
  - Bevestigings emails, password resetten, etc.
- Aanpassen is niet moeilijk:
  - Overerf van `IdentityUser`
  - Pas de front-end aan in de Razor Pages

INTERMEZZO: RAZOR PAGES

- Razor Pages: minder flexibiliteit, maar georganiseerdere code
- Razor Pages: Lijkt erg op MVC
- MVC en Razor Pages kunnen beide naast elkaar bestaan

INTERMEZZO: RAZOR PAGES

- Razor Pages: minder flexibiliteit, maar georganiseerdere code
- Razor Pages: Lijkt erg op MVC
- MVC en Razor Pages kunnen beide naast elkaar bestaan

INTERMEZZO: RAZOR PAGES

- Razor Pages: minder flexibiliteit, maar georganiseerdere code
- Razor Pages: Lijkt erg op MVC
- MVC en Razor Pages kunnen beide naast elkaar bestaan

EXTERNAL LOGINS

Volg de guides op MSDN:

- Google
- Facebook
- Microsoft
- Twitter

of LinkedIn, Instagram, Reddit, Github, Yahoo, Tumblr, Pinterest, Pocket, Flickr, Dribble, Vimeo, SoundCloud

AUTHORIZATION

- Niet ingelogd/ingelogd: [Authorize] en [AllowAnonymous]
- Geavanceerder: rollen
  - Gebruik 'dependency injection' voor `UserManager<InlogbaarUser>` en `RoleManager<IdentityRole>`.
  - In IdentityHostingStartup.cs (dus niet Startup.cs!):

    ```
    services.AddIdentity<InlogbaarUser, IdentityRole>
        (options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<InlogbaarContext>()
        .AddRoles<IdentityRole>()
        .AddRoleManager<RoleManager<IdentityRole>>();
    ```

  - Voorbeeld-gebruik:

    ```
    [Authorize(Roles = "GroteBaas,AfdelingFinancien")]
    public class SalarisController : Controller { }
    ```

    en

    ```
    await roleManager.CreateAsync(new IdentityRole { Name = "GroteBaas" });
    await userManager.AddToRoleAsync(user, "GroteBaas");
    ```

- Georganiseerder: policies.
- Nog geavanceerder: custom authorize attributen.

AUTHORIZATION

AUTHORIZATION

AUTHORIZATION

AUTHORIZATION

AUTHORIZATION

AUTHORIZATION

OWASP TOP 10

Zie hier. Wat weten we nu?

- **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
- **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- **XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
- **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP TOP 10

Zie hier. Wat weten we nu?

- **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.
- **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- **XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.
- **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP TOP 10

Zie hier. Wat weten we nu?

- **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.
- **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- **XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.
- **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP TOP 10

Zie hier. Wat weten we nu?

- **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.
- **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- **XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.
- **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP TOP 10

Zie hier. Wat weten we nu?

- **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.
- **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- **XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.
- **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP TOP 10

Zie hier. Wat weten we nu?

- **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
- **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- **XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
- **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP TOP 10

Zie hier. Wat weten we nu?

- **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.
- **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- **XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.
- **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP TOP 10

Zie hier. Wat weten we nu?

- ⚔**Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- ⚔**Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.
- ⚔**Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- ⚔**XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- ⚔**Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.
- ⚔**Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- ⚔**Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- ⚔**Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- ⚔**Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- ⚔**Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP TOP 10

Zie hier. Wat weten we nu?

- **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.
- **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- **XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.
- **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP TOP 10

Zie hier. Wat weten we nu?

- **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.
- **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- **XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.
- **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP TOP 10

Zie hier. Wat weten we nu?

- **Injection**. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
- **Broken Authentication**. Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.
- **Sensitive Data Exposure**. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
- **XML External Entities** (XXE). Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
- **Broken Access Control**. Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users’ accounts, view sensitive files, modify other users’ data, change access rights, etc.
- **Security Misconfiguration**. Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.
- **Cross-Site Scripting XSS**. XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
- **Insecure Deserialization**. Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
- **Using Components with Known Vulnerabilities**. Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
- **Insufficient Logging & Monitoring**. Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OPDRACHT