Web Application Security

Lecture 4

Tamara Rezk

Web Application Security

- 19/11: Historical introduction to technologies and vulnerabilities that accompany them
- 26/11: Defenses XSS and DOM-XSS: sanitizers, Content Security Policy
- 10/12: Same Origin Policy
- 17/12: JavaScript vulnerabilities and defences
- 07/01: Workshop and Pysa
- 14/01: Workshop and Trusted Types (Google)
- 28/01: examen (9h -12h30), required: mandatory TP exercises

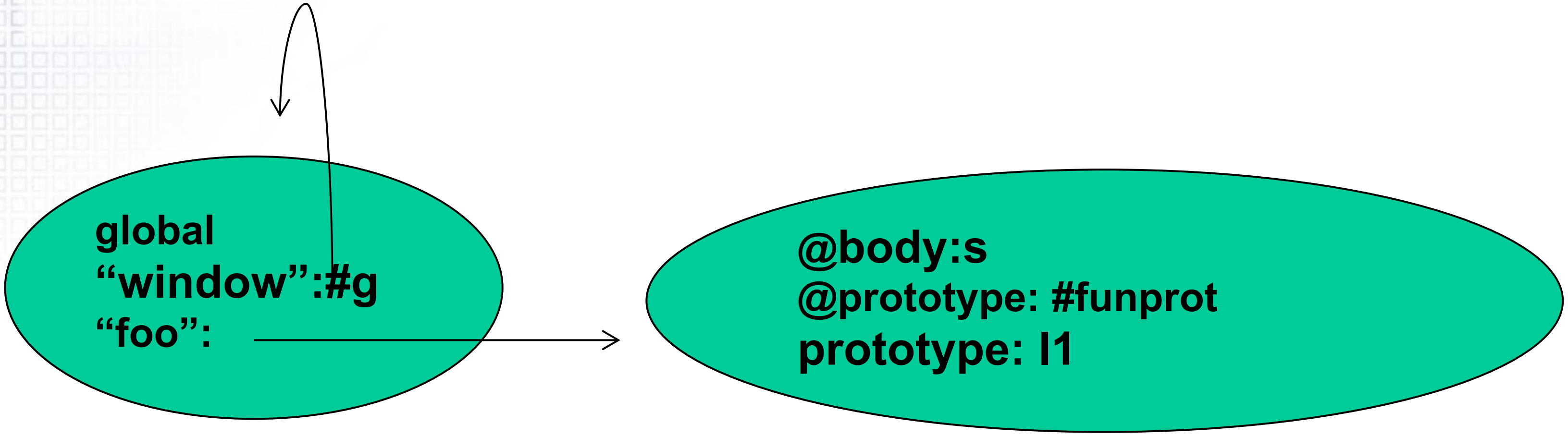
Workshops

- Workshop presentations 7/1 and 14/1
  - 15’ presentation
  - 5’ questions : choose one group and prepare 2 questions.
  - write your topic in the excel sheet and also the questions group
  - workshop is mandatory so please justify absence
- Google presentation on Trusted types: 14/1 1hour (topics included in exam questions, cf. DOM-XSS)

Functions

```
function foo(x) { s} ;
```

- a function is also a property of an object (global in this example)
- when a function is defined, it is NOT called (or executed).
- notice that a function definition IS an object



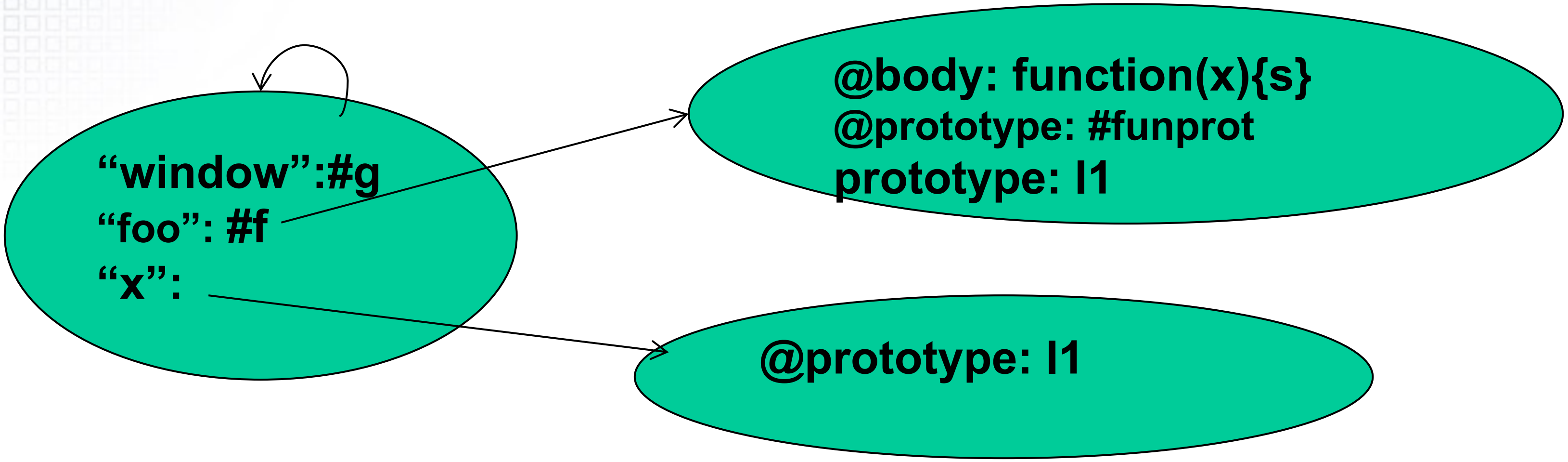
Example of a JavaScript function

```
var  count=0;
function
   increment(inc){
       var count=5;
        if (inc ==
   undefined)
             {inc = 1;}
        count += inc;
        return count;
}
```

Functions as Constructors

```
function foo(x) ={s} ;
var x = new foo(3);
```

- when a new function instance is created, it IS called (as constructor)
- IMPORTANT DETAIL: “this” is now associated to the new object
- it is executed in the same scope where the object was defined.
- new is a way to set the internal @prototype



“window”:#g
“foo”: #f
“x”:

@body: function(x){s}
@prototype: #funprot
prototype: l1

@prototype: l1

Example with this

```
var  count=0;
var increment=
function(inc) {
     var count=5;
      if (inc ==
  undefined)
  {inc = 1;}
   this.count += inc;
  console.log(this.count);
}
```

Example with this

```
var  count=0;
var increment=
function(inc) {
     var count=5;
      if (inc ==
  undefined)
  {inc = 1;}
   this.count += inc;
  console.log(this.count);
}
```

Executing

```
new increment();
```

prints

NaN (not a number!)

because “count” is not defined in the new object.

Example with this: correct form

```
var  count=0;
var increment=
function(inc) {
     this.count=5;
      if (inc == undefined)
  {inc = 1;}
   this.count += inc;
   console.log(this.count);
}
```

Executing

`new increment();`

returns a new object where property count is equal to 6

Invocation of a function

```
increment();
```

```
increment(2);
```

Function: what's the difference?

with this

```
var  count=0;
function
  increment(inc) {
     var count=5;
      if (inc ==
  undefined)
       {inc = 1;};
      this.count +=
  inc;
      return
  this.count;
}
```

without this

```
var  count=0;
function increment
  (inc) {
     var count=5;
      if (inc ==
  undefined)
       {inc = 1;};
      count += inc;
      return count;
}
```

Function: what’s the difference?

with this

```
var  count=0;
function
  increment(inc) {
     var count=5;
      if (inc ==
  undefined)
       {inc = 1;};
      this.count +=
  inc;
      return
  this.count;
}
```

without this

```
var  count=0;
function increment
   (inc) {
        var count=5;
         if (inc ==
  undefined)
         {inc = 1;};
         count += inc;
         return count;
}
```

this is bound to the global object

Function: this is bound to window

Important detail:

When a function that returns “this” is called as a function and not as a constructor, this is bound to the global object (window)

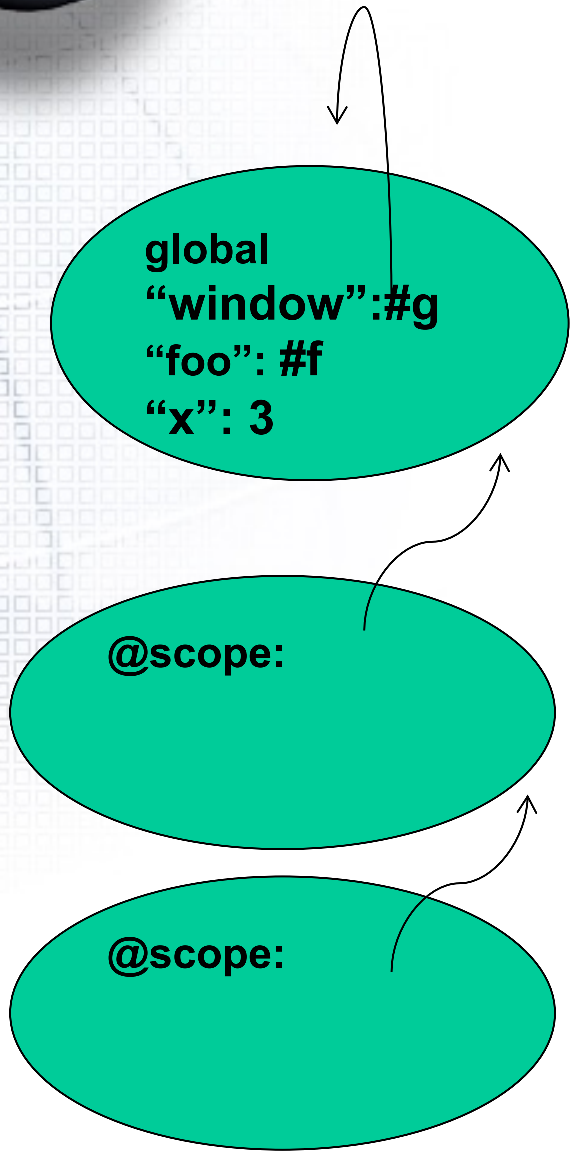
Security problem:

If a function that returns this is used by an attacker, the attacker has access to all resources in the page that are linked to window (in particular document and document.cookie). Solution: don’t expose window

JavaScript

LEAKS VIA SCOPE

Scope Chain



```
var x=3;
function foo (){console.log(x)};
```

Example using scope chain

```
function Foo() {
var x;
x =3 ;
y = x;
}

function Bar() {
y = x;
x = x +1;
}

var x = 0;
var y = 0;
Foo();
Bar();
```

What’s the value of
global x and y after
Foo ()?
and after Bar()?

Leaks via Scope

Important detail:

When a variable is not local to the object, then JavaScript mounts the scope chain to look for the variable (analogous detail for properties in the prototype chain).

Security problem:

- Integrity: an attacker can write a variable higher up in the scope chain
- Confidentiality: an attacker can read a variable higher up in the scope chain

Solution: use “var” , isolate code, be aware of untrusted “scope”

(analogous problems for properties in the prototype chain).

JavaScript

LEAKS VIA IMPLICIT TOSTRING

Is this function safe?

```
lookup =
function(o, prop) {
  if (prop === “secretproperty")
  {
    return "unsafe!"; }
  else {
    return o[prop]; } }
```

Is this function safe?

```
lookup =
function(o, prop) {
  if (prop === “secretproperty")
  {
    return "unsafe!"; }
  else {
    return o[prop]; } }
```

If `prop` is not a string, JavaScript invokes the `.toString` method to convert the value to a string

…in fact, `lookup` is unsafe!

```
badObj =
  {toString:
    function () {
         return "secretproperty"}}

  lookup(window, badObj)
➔ window[badObj]
➔ window[{toString: …}]
➔ window[{toS…: …}.toS… ()]
➔ window[(function () …) ()]
➔ window["secretproperty"]
```

Leaks via implicit toString invocation

Important detail:

In JavaScript `o.f` is treated as `o["f"]`

Security problem:

Via the implicit invocation of toString, a property could evaluate to an undesirable choice of the attacker

JavaScript

LEAKS VIA EVAL

The eval that men do

This is a string

```
eval(
   “function attackercode(){…};attackercode();}”
    )
```

More evals: e.g., setTimeout:

```
function f() { alert('hello'); }
setTimeout(f, 1000);
```

```
var s = "alert('hello') ";
setTimeout(s, 1000);
```

example setTimeout

Any JavaScript string!

Leaks via eval

Important detail:

Eval interprets any string as code

Security problem:

If a string of the attacker gets to eval,

attacker executes his code

JavaScript

LEAKS VIA NATIVE FUNCTIONS

Example with setTimeout: what is the result of executing this code?

```
<script>
function fac(x) {
    if (x <= 1) {
        return 1;
    }
    return x*fac(x-
1);
}
r = fac(3);
s = "alert("+r+")"
setTimeout(s, 100)
</script>
```

example setTimeout

What happens now?

```
<script src=somecode.js></script>
<script>
function fac(x) {
    if (x <= 1) {
        return 1;
    }
    return x*fac(x-1);
}
r = fac(4);
s = "alert("+r+")"
setTimeout(s, 100)
</script>
```

Leaks via native functions

Important detail:

Native functions code can be rewritten

Security problem:

If attacker rewrites code of native function, when trusted code calls a native function, it is executing code of attacker! Solutions: use “const” , freeze objects, isolated untrusted code ...

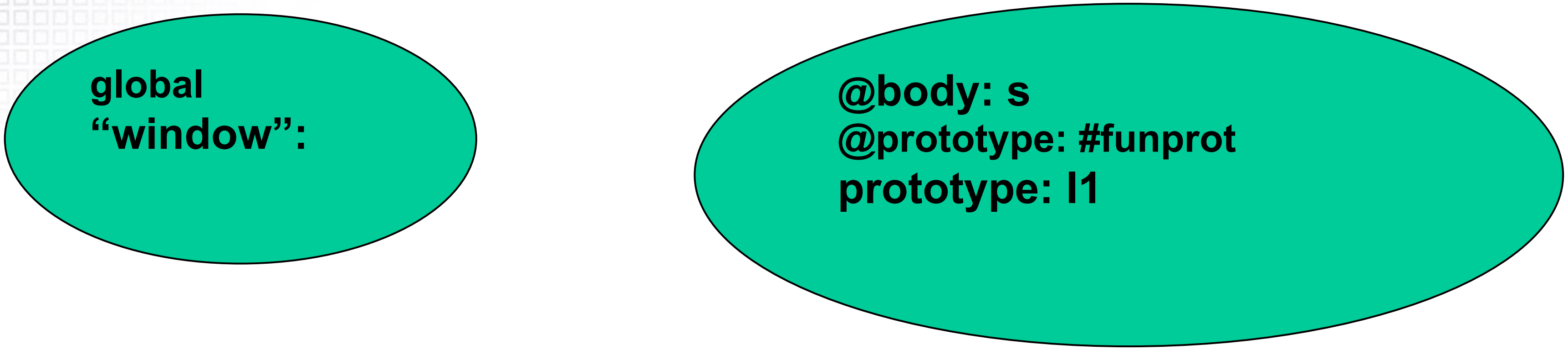
JavaScript

ISOLATION WITHOUT SOP

Anonymous Functions

```
function(x){s};
```

- anonymous functions cannot be reached via the scope object (window for example)



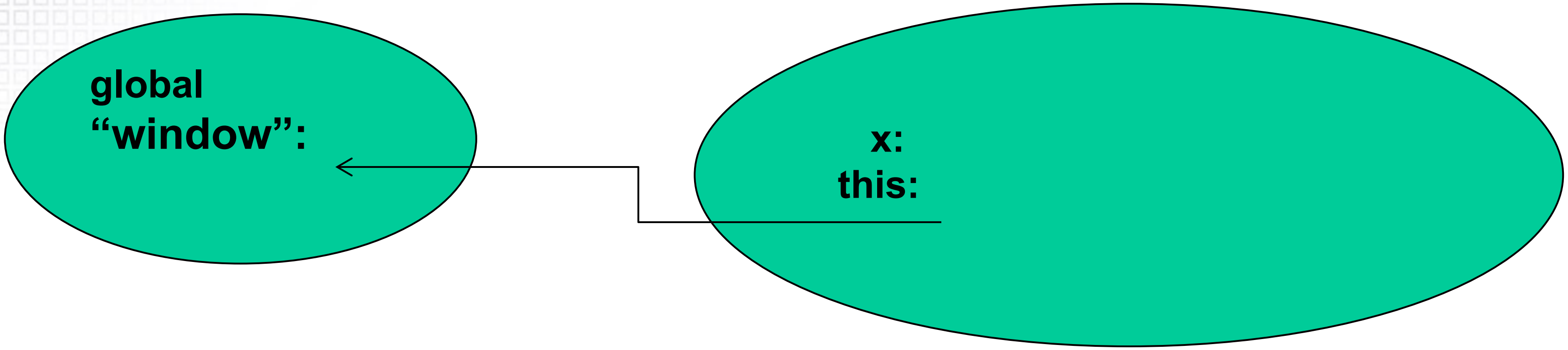
global
“window”:

@body: s
@prototype: #funprot
prototype: l1

Anonymous Functions

```
function(x){s};
```

at execution: this points to global as with named functions



Example Anonymous Function

- If you want to execute code only once, in isolation from attacker, you can use anonymous functions:

```
<script src= attacker.com/code.js></script>
<script>
 (function(){    some private code }  ())
</script>
```

Attacker cannot access memory of the anonymous function. But code is still vulnerable if it uses native functions

Example Anonymous Function

- If you want to define an api and expose it to attacker

```
<script src=  attacker.com/code.js></script>
<script>
  api = (function(){
          some private code
          return function(){some public code}
              } ())
</script>
```

Example Anonymous vs Named

```
function Foo(){
          var bar = 0; …
          return bar;};
result = Foo();
```

vs

```
result= (function(){
          var bar = 0; …
          return bar
               } ())
```

Example Anonymous vs Named

```
function Foo(){
         var bar = 0; …
         return bar; };
result = Foo();
```

You can use Foo.toString()

VS

```
result= (function(){
         var bar = 0; …
         return bar
              } ())
```

Dedicated Isolated Mechanisms:
SES , isolated-vm (server), same origin policy (client), etc

Anonymous functions

Important detail:

They are not linked to the global object

Security solution?

They help to encapsulate state but attacker can still read its code via XMLHttpRequest() if SOP does not, for example. Code of named functions can be read with toString.

General JavaScript security measures

- Do not expose “window” to untrusted code
  - Untrusted code: any code coming from another server
  - Untrusted code: any code coming from the client
- Watch out for implicit type coercions (like calls to toString)
- Avoid evals (explicit or implicit)
- Make sure that native functions execute original code
- JavaScript isolation: SES, isolated-vm, same origin policy, etc.

Today

- Checking Path Traversal and Guestbook in priority (approx. 1/3 of students)

- Answering questions on exercises on JavaScript

- Deadline to provide topics for the workshop

7/1

- Install Pysa before the course

https://github.com/facebook/pyre-check/

- OR download the virtual machine I will leave on the slack.

TP

1. See code for boot.js, trusted.js, mashup1.js. Without changing this code, write code for attacker.js in order to make trusted.js execute unwanted code.
2. Change boot.js (and if need be trusted.js) to avoid the attack

TP

3. Let adapi.js be the code for some external gadget. Assume that code for adapi.js has been verified and cannot access window directly, however it

- has access to function integrator whenever it is available. For each of the following
- versions of the mashup, can the external gadget adapi.js
- read the value of secret?
- obtain a pointer to window?

TP

V1

```
< script >
function integrator(){secret = 42; return this;}
</script >
< script src = http : //adserver.com/adapi.js >
```

TP

V2

```
< script >
function integrator(){var secret = 42; return this;}
</script >
< script src = http : //adserver:com=adapi.js >
```

TP

V3

```
< script >
(function (){secret = 42; return this;})()
</script >
< script src = http : //adserver:com=adapi.js >
```

TP

4. Assume you have a function lookup that will replace any access to a property of the form o[prop] in attacker code by lookup(o, prop). The goal of lookup is to prevent any access to a special property “secretproperty”. Which of the following 2 implementations of lookup satisfy this goal? Justify your answer.

TP

V1

```
lookup1 =
function(o, prop){
if (prop === ‘secretproperty’){
return "unsafe!"; }
else {
return o[prop]; }
```

TP

V2

```
lookup2 =
function(o, prop){
var goodprop = {
‘publicproperty’: ‘publicproperty’,
‘secretproperty’: ‘publicproperty’}[prop];
return o[goodprop]; }
```