Security Types for Web Applications

Antoine Delignat-Lavaud

Under the supervision of S. Maffeis and K. Bhargavan
PROSECCO, INRIA Paris-Rocquencourt

September 3, 2012

Introduction: Our Goals

Security of Web Applications

- Application logic shared between web server and browser client.
- Complex interaction over HTTP between at least 2 main principals, often more.
- Other interactions between client / server and third parties.
- Security goals: confidentiality and integrity of communication, authentication, data access control, sharing...
- Use of cryptography to achieve these goals.

Introduction: Our Goals

Security of Web Applications

- Application logic shared between web server and browser client.
- Complex interaction over HTTP between at least 2 main principals, often more.
- Other interactions between client / server and third parties.
- Security goals: confidentiality and integrity of communication, authentication, data access control, sharing...
- Use of cryptography to achieve these goals.

Introduction: Our Goals

Security of Web Applications

- Application logic shared between web server and browser client.
- Complex interaction over HTTP between at least 2 main principals, often more.
- Other interactions between client / server and third parties.
- Security goals: confidentiality and integrity of communication, authentication, data access control, sharing...
- Use of cryptography to achieve these goals.

Introduction: Our Goals

Security of Web Applications

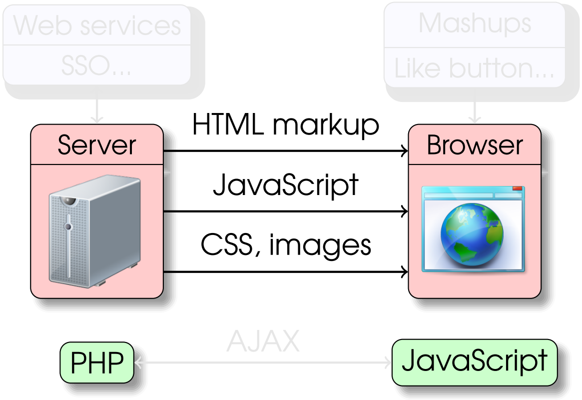
- Application logic shared between web server and browser client.
- Complex interaction over HTTP between at least 2 main principals, often more.
- Other interactions between client / server and third parties.
- Security goals: confidentiality and integrity of communication, authentication, data access control, sharing...
- Use of cryptography to achieve these goals.

Introduction: Our Goals

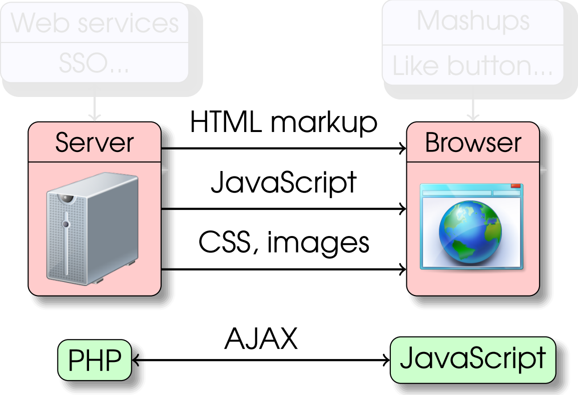
Security of Web Applications

- Application logic shared between web server and browser client.
- Complex interaction over HTTP between at least 2 main principals, often more.
- Other interactions between client / server and third parties.
- Security goals: confidentiality and integrity of communication, authentication, data access control, sharing...
- Use of cryptography to achieve these goals.

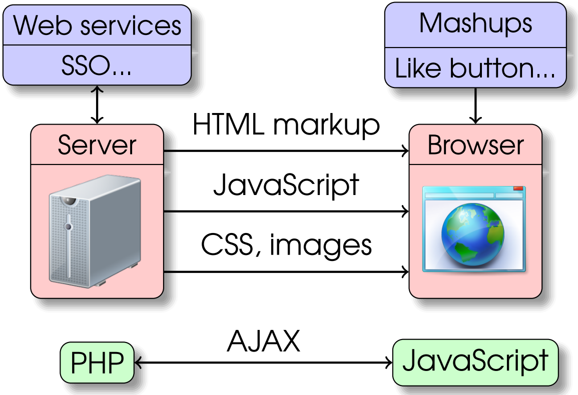
Web application overview



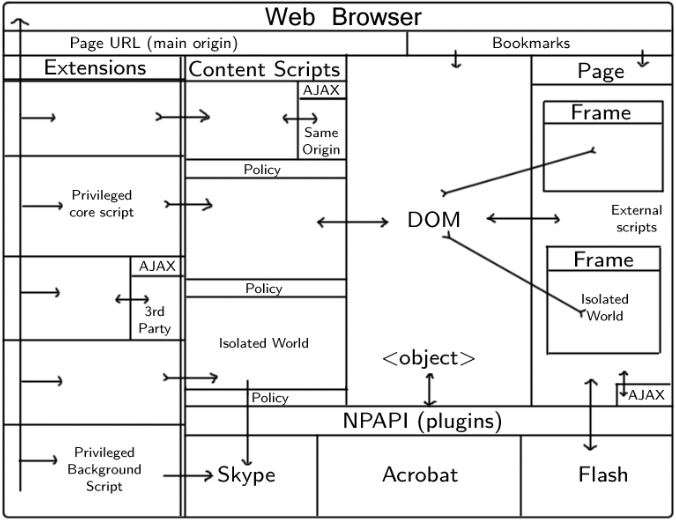
Web application overview



Web application overview



Browser security



Our contribution

- We focus our attention on the client-side interactions.
- We conducted a review on the security of host-proof web applications and found a variety of attack vectors.
- We investigated the problem of loading trusted JavaScript code into an untrusted environment.
- We propose a subset of JavaScript we belive is safe to use in such environments.
- We implemented a type system able to check if a given script belongs to that subset.

Our contribution

- We focus our attention on the client-side interactions.
- We conducted a review on the security of host-proof web applications and found a variety of attack vectors.
- We investigated the problem of loading trusted JavaScript code into an untrusted environment.
- We propose a subset of JavaScript we belive is safe to use in such environments.
- We implemented a type system able to check if a given script belongs to that subset.

Our contribution

- We focus our attention on the client-side interactions.
- We conducted a review on the security of host-proof web applications and found a variety of attack vectors.
- We investigated the problem of loading trusted JavaScript code into an untrusted environment.
- We propose a subset of JavaScript we belive is safe to use in such environments.
- We implemented a type system able to check if a given script belongs to that subset.

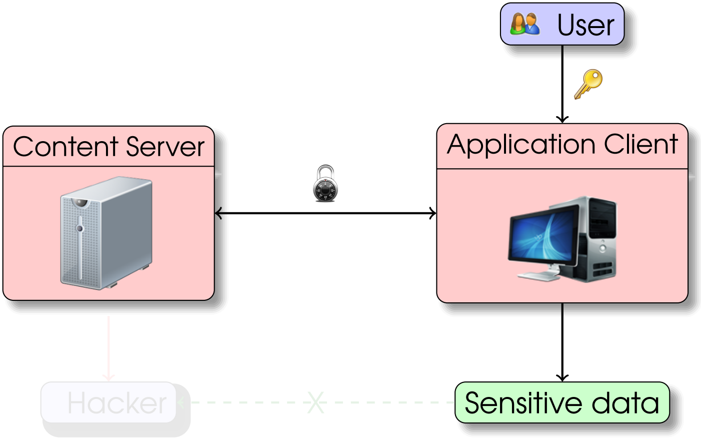
Our contribution

- We focus our attention on the client-side interactions.
- We conducted a review on the security of host-proof web applications and found a variety of attack vectors.
- We investigated the problem of loading trusted JavaScript code into an untrusted environment.
- We propose a subset of JavaScript we belive is safe to use in such environments.
- We implemented a type system able to check if a given script belongs to that subset.

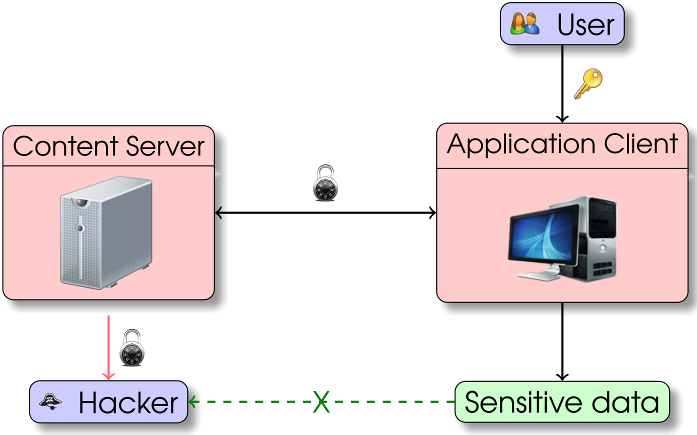
Our contribution

- We focus our attention on the client-side interactions.
- We conducted a review on the security of host-proof web applications and found a variety of attack vectors.
- We investigated the problem of loading trusted JavaScript code into an untrusted environment.
- We propose a subset of JavaScript we belive is safe to use in such environments.
- We implemented a type system able to check if a given script belongs to that subset.

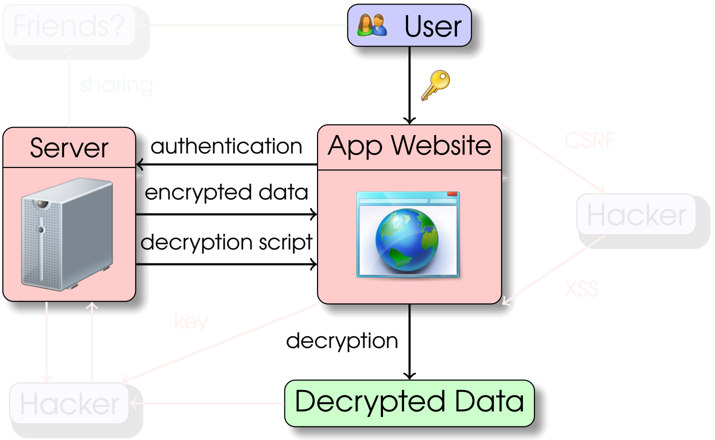
Host-Proof Application Design



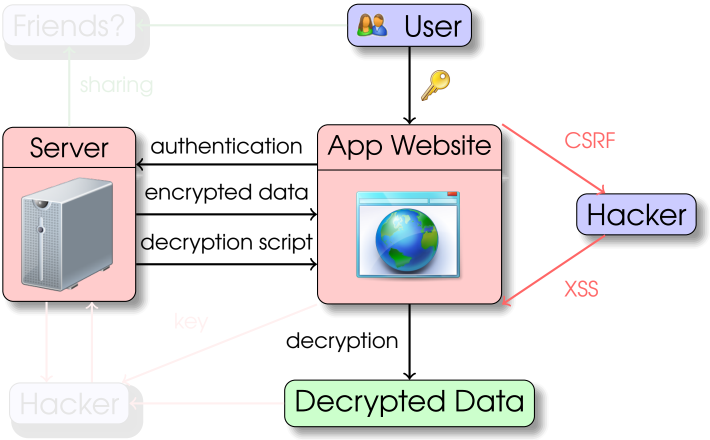
Host-Proof Application Design



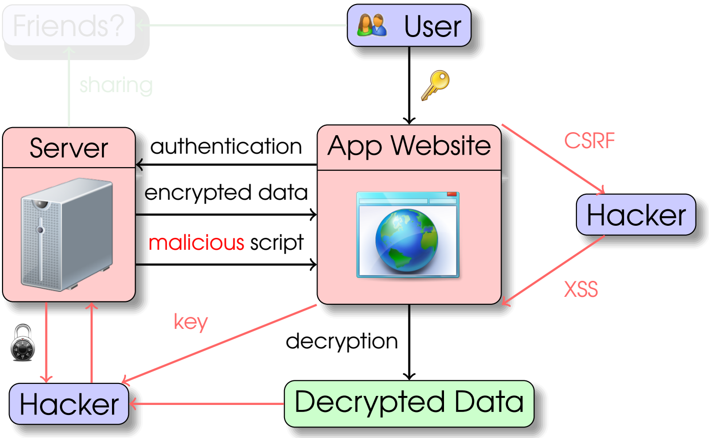
Encrypted cloud storage



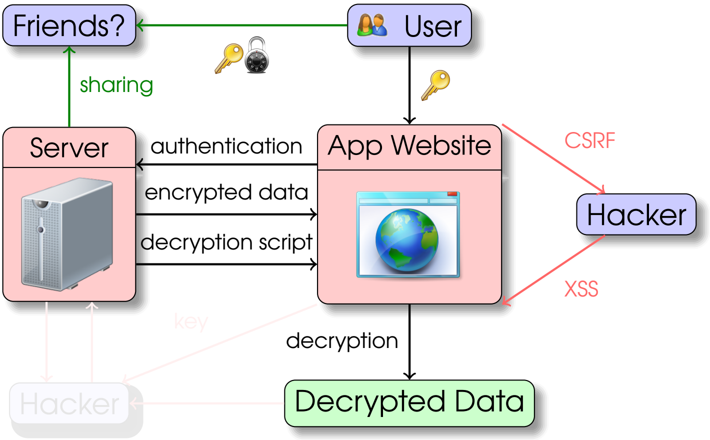
Encrypted cloud storage



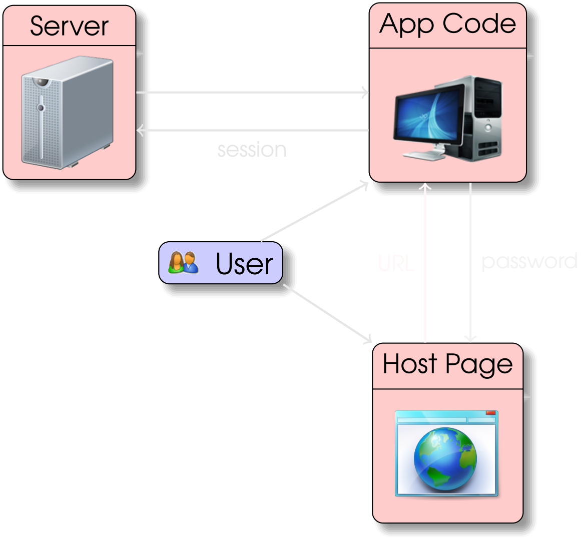
Encrypted cloud storage



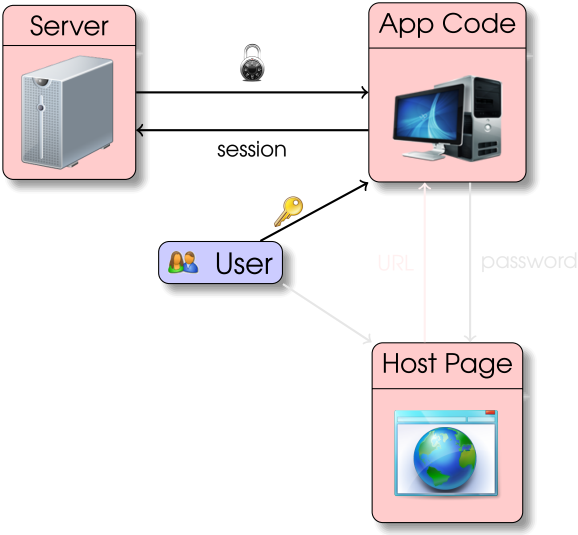
Encrypted cloud storage



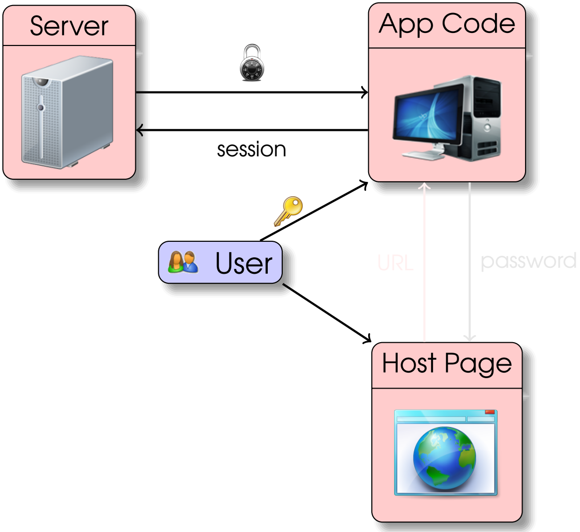
Password Manager Browser Extension



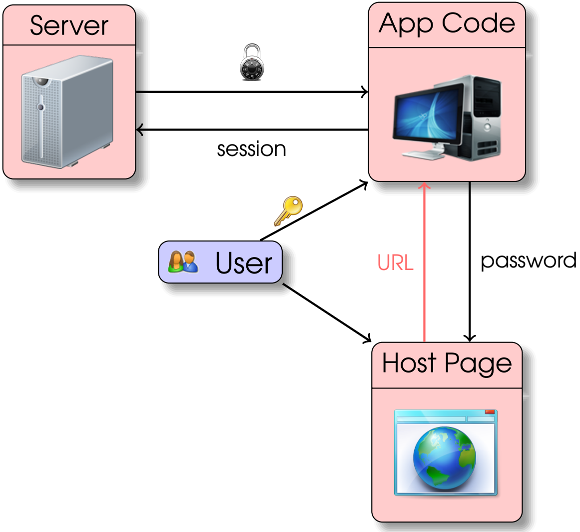
Password Manager Browser Extension



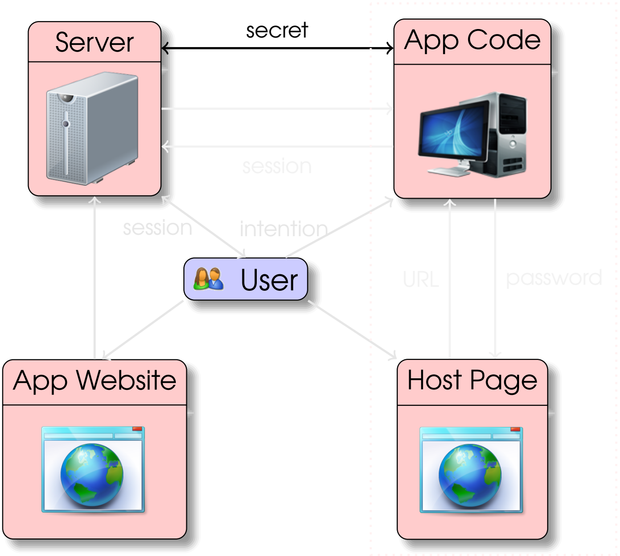
Password Manager Browser Extension



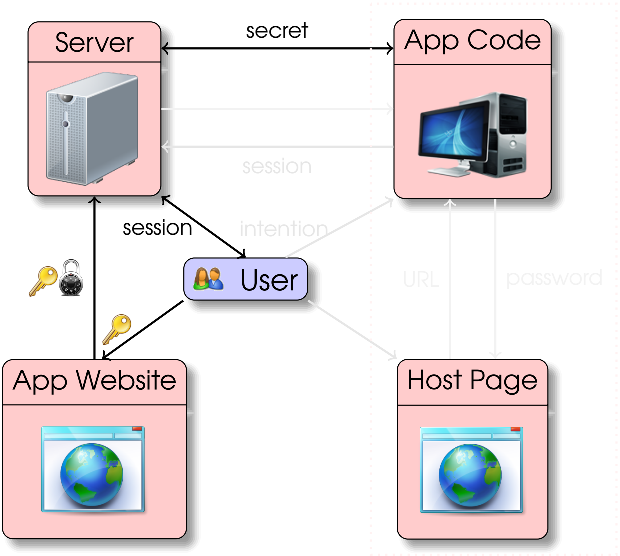
Password Manager Browser Extension



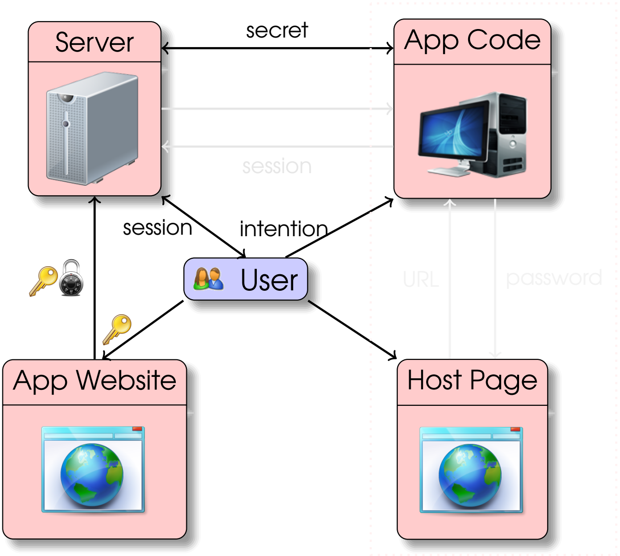
Password Manager Bookmarklet



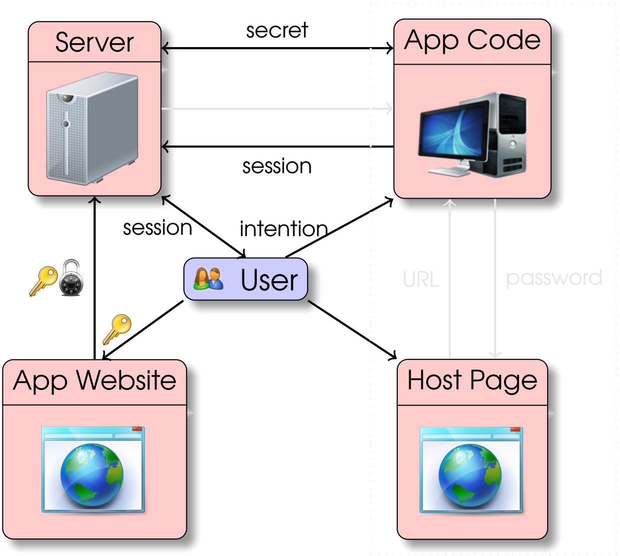
Password Manager Bookmarklet



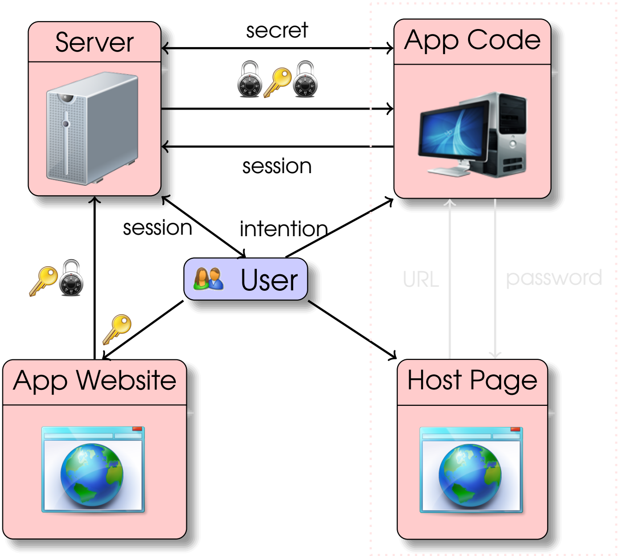
Password Manager Bookmarklet



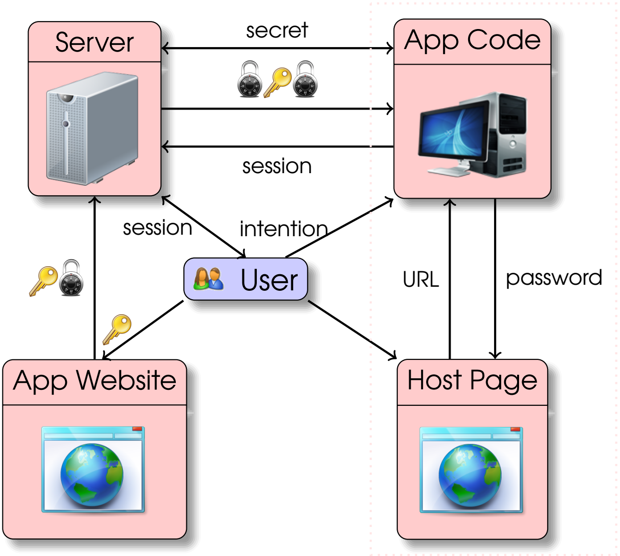
Password Manager Bookmarklet



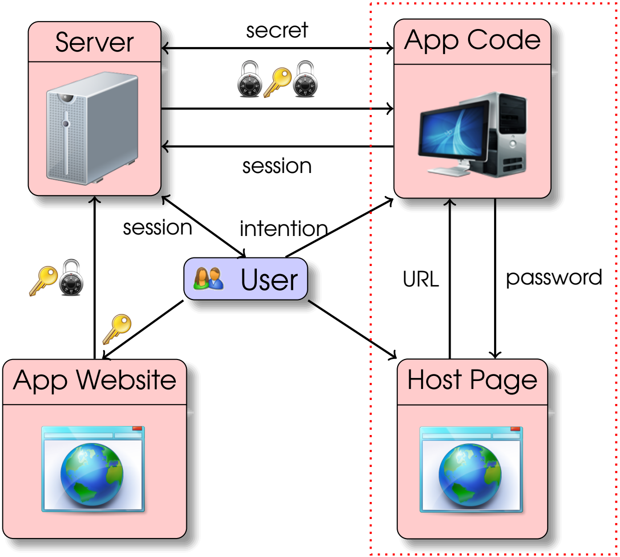
Password Manager Bookmarklet



Password Manager Bookmarklet



Password Manager Bookmarklet



Attacks

What can go wrong?

- Incorrect use of crypto.
- Usual web attacks (XSS/CSRF).
- No data/code separation.
- Bad key management.

Attacks

What can go wrong?

- Incorrect use of crypto.
- Usual web attacks (XSS/CSRF).
- No data/code separation.
- Bad key management.

Attacks

What can go wrong?

- Incorrect use of crypto.
- Usual web attacks (XSS/CSRF).
- No data/code separation.
- Bad key management.

Attacks

What can go wrong?

- Incorrect use of crypto.
- Usual web attacks (XSS/CSRF).
- No data/code separation.
- Bad key management.

No ciphertext integrity protection

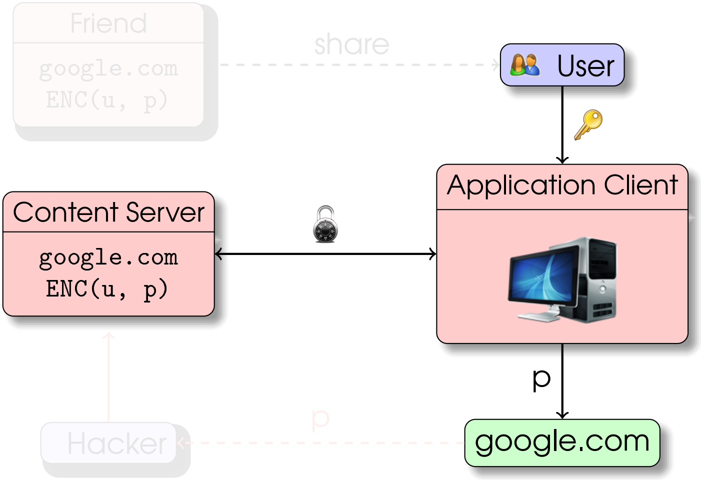
RoboForm Passcard

`URL3:Encode(URL)`
`+PROTECTED-2+`
`<`ENC_k`(username,password)>`

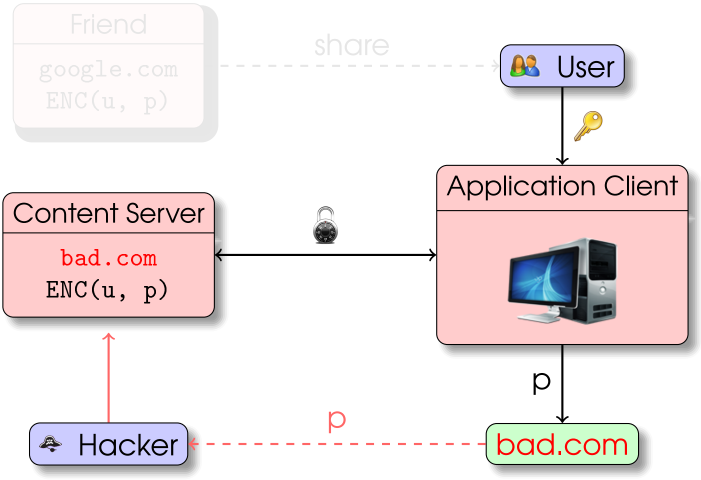
1Password Keychain

`{"uuid":...,"title":..., "location":URL,`
`"encrypted":<`ENC_k`(username,password)>}`

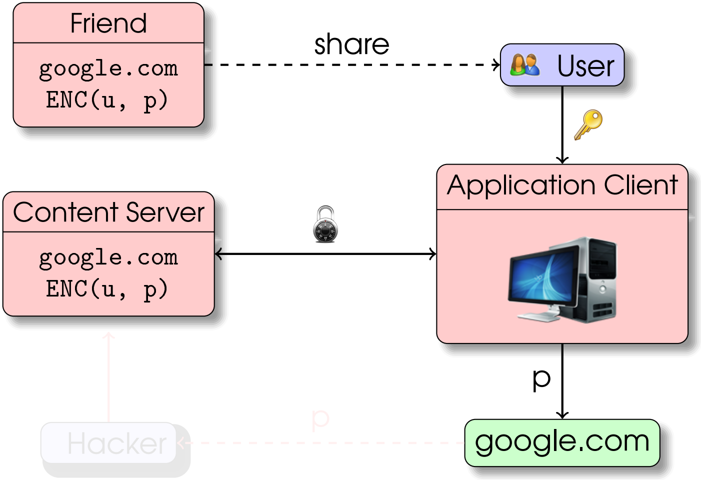
No ciphertext integrity protection



No ciphertext integrity protection



No ciphertext integrity protection



Classic problem: URL authenticating

- Browser extension-based password managers;
- Match URL with password database in JS.
- Error-prone RegExp matching.

parseUri pattern

```
/^(?:([^:\/?#]+):)?(?:\/\/((?:(([^:@]*)
(?::([^:@]*))?)?@)?([^:\/?#]*)(?::(\d*))?))?
((((?:[^?#\/]*\/)*)([^?#]*))(?:\?([^#]*))?
(?:#(.*))?)/
```

Incorrect

`http://bad.com/#@accounts.google.com`

Classic problem: URL authenticating

- Browser extension-based password managers;
- Match URL with password database in JS.
- Error-prone RegExp matching.

parseUri pattern

```
/^(?:([^:\/?#]+):)?(?:\/\/((?:(([^:@]*)
(?::([^:@]*))?)?@)?([^:\/?#]*)(?::(\d*))?))?
((((?:[^?#\/]*\/)*)([^?#]*))(?:\?([^#]*))?
(?:#(.*))?)/
```

Incorrect

```
http://bad.com/#@accounts.google.com
```

Classic problem: URL authenticating

- Browser extension-based password managers;
- Match URL with password database in JS.
- Error-prone RegExp matching.

parseUri pattern

```
/^(?:([^:\/?#]+):)?(?:\/\/((?:(([^:@]*)
(?::([^:@]*))?)?@)?([^:\/?#]*)(?::(\d*))?))?
((((?:[^?#\/]*\/)*)([^?#]*))(?:\?([^#]*))?
(?:#(.*))?)/
```

Incorrect

`http://bad.com/#@accounts.google.com`

Classic problem: URL authenticating

- Browser extension-based password managers;
- Match URL with password database in JS.
- Error-prone RegExp matching.

parseUri pattern

```
/^(?:([^:\/?#]+):)?(?:\/\/((?:(([^:@]*)
(?::([^:@]*))?)?@)?([^:\/?#]*)(?::(\d*))?))?
((((?:[^?#\/]*\/)*)([^?#]*))(?:\?([^#]*))?
(?:#(.*))?)/
```

Incorrect

```
http://bad.com/#@accounts.google.com
```

Classic problem: URL authenticating

- Browser extension-based password managers;
- Match URL with password database in JS.
- Error-prone RegExp matching.

parseUri pattern

```
/^(?:([^:\/?#]+):)?(?:\/\/((?:(([^:@]*)
(?::([^:@]*))?)?@)?([^:\/?#]*)(?::(\d*))?))?
((((?:[^?#\/]*\/)*)([^?#]*))(?:\?([^#]*))?
(?:#(.*))?)/
```

Incorrect

```
http://bad.com/#@accounts.google.com
```

Fishing attack on 1Password extension

URL parsing code

```
var href = getBrowser().contentWindow
                  .location.href + "/";
var domain = href.replace(
        /^http[s]*:\/\/(.*?)\/.*$/i, "$1");
var middle = domain.replace(
                  /^(www.)*(.*)/i, "$2");

return middle.substring(0,1).toUpperCase() +
       middle.substring(1,middle.length);
```

Fishing URL

```
http://www.google.com:xxx@bad.com
```

Fishing attack on 1Password extension

URL parsing code

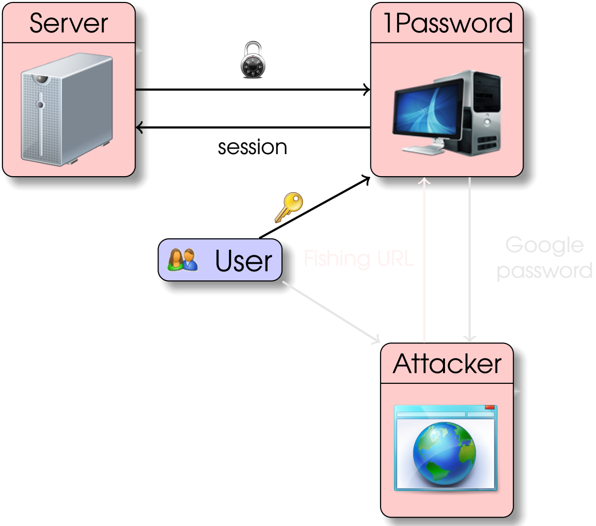
```
var href = getBrowser().contentWindow
                 .location.href + "/";
var domain = href.replace(
        /^http[s]*:\/\/(.*?)\/.*$/i, "$1");
var middle = domain.replace(
                  /^(www.)*(.*)/i, "$2");

return middle.substring(0,1).toUpperCase() +
       middle.substring(1,middle.length);
```

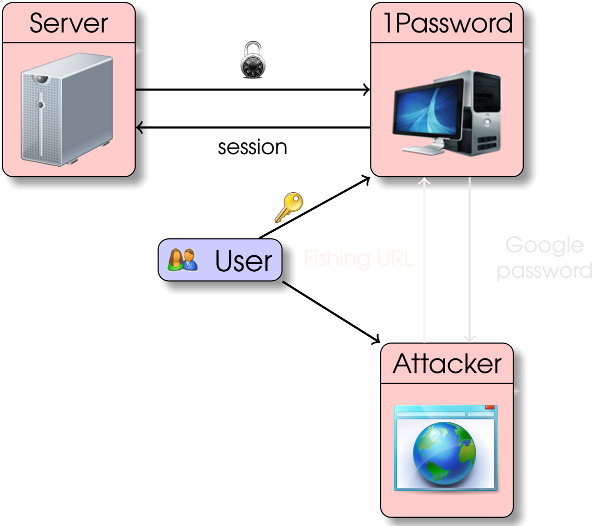
Fishing URL

```
http://www.google.com:xxx@bad.com
```

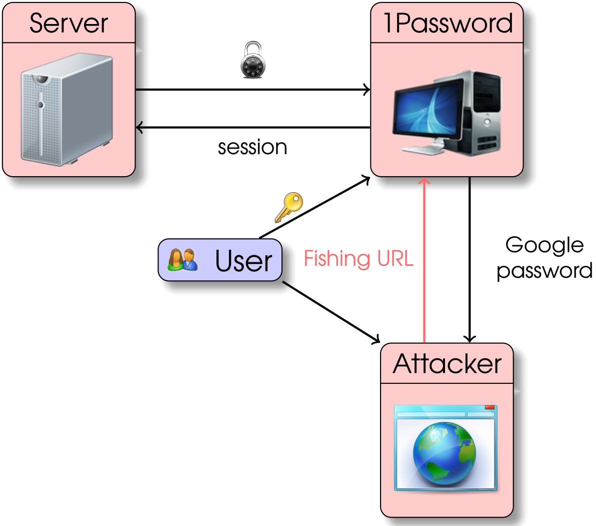
1Password fishing attack



1Password fishing attack



1Password fishing attack



Code/data separation

Web interfaces

- Hard to maintain client-side decryption due to Javascript limitations.
- Login form exposed to web attacks.
- Decryption in the same scope as various GUI and user data.

Code/data separation

Web interfaces

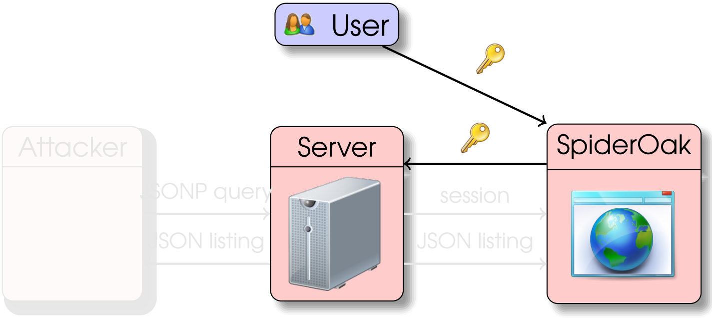
- Hard to maintain client-side decryption due to Javascript limitations.
- Login form exposed to web attacks.
- Decryption in the same scope as various GUI and user data.

Code/data separation

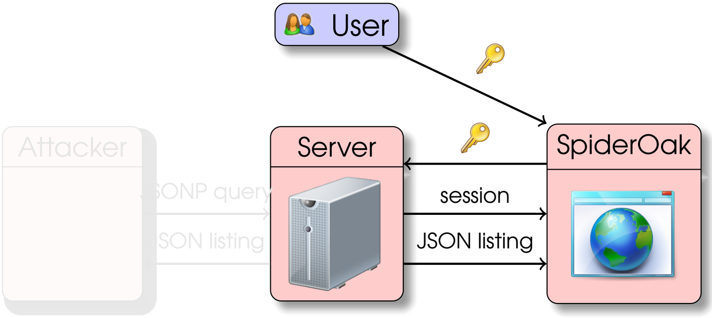
Web interfaces

- Hard to maintain client-side decryption due to Javascript limitations.
- Login form exposed to web attacks.
- Decryption in the same scope as various GUI and user data.

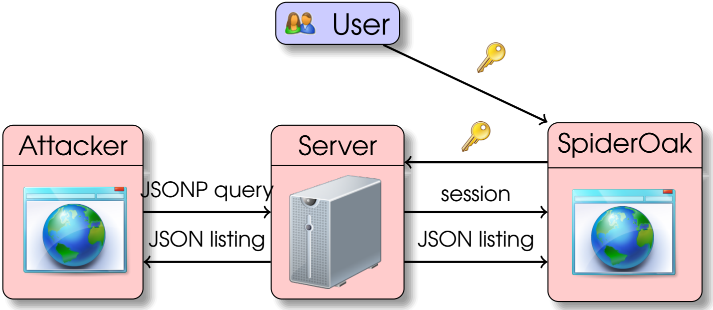
SpiderOak


User
Server
SpiderOak
Attacker
JSONP query
JSON listing
session
JSON listing

SpiderOak


User
Attacker
JSONP query
JSON listing
Server
session
JSON listing
SpiderOak

SpiderOak



User
Attacker
Server
SpiderOak
JSONP query
JSON listing
session
JSON listing

SpiderOak

Query

```
https://spideroak.com/storage/<u32>/?callback=f
```

Result

```
f({
 "stats": {
   "firstname": "...",
   "lastname": "...",
   "devices": ...,
 },
 "devices": [
  ["pc1", "pc1/"],["laptop", "laptop/"],...
 ]
})
```

SpiderOak

Query

```
https://spideroak.com/storage/<u32>/shares
```

Result

```
{
"share_rooms" : [
  "url" : "/browse/share/<id>/<key>",
  "room_key" : "<key>",
  "room_description" : "" ,
  "room_name": "<room>"
 ],
 "share_id" : "<id>",
 "share_id_b32" : "<u32>"
}
```

Key management

A difficult challenge

- All applications implement some form of sharing.
- Full database vs per-entry dilemma.
- Bias towards features rather than security.

Key management

A difficult challenge

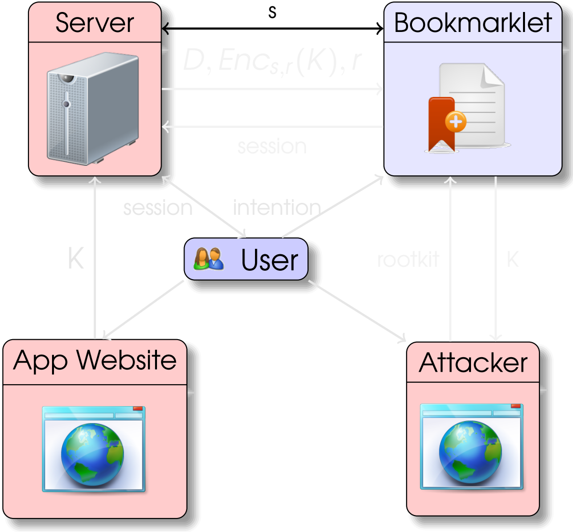
- All applications implement some form of sharing.
- Full database vs per-entry dilemma.
- Bias towards features rather than security.

Key management

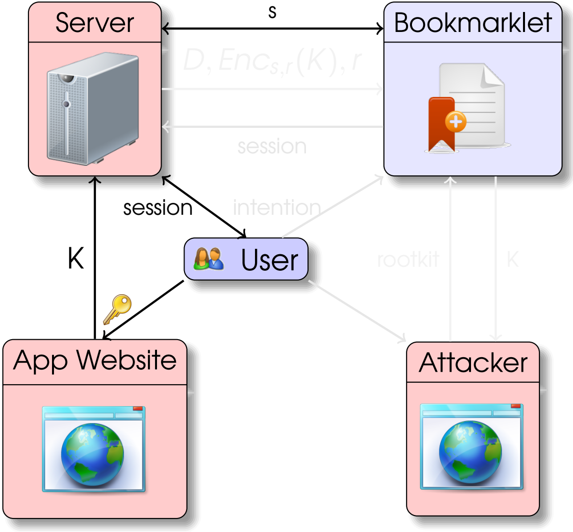
A difficult challenge

- All applications implement some form of sharing.
- Full database vs per-entry dilemma.
- Bias towards features rather than security.

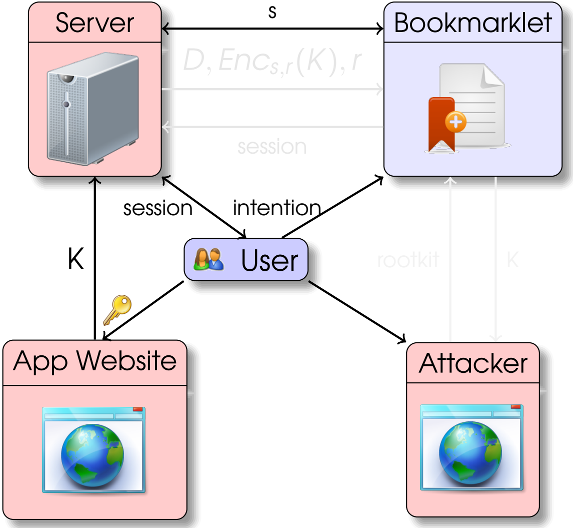
LastPass login bookmarklet



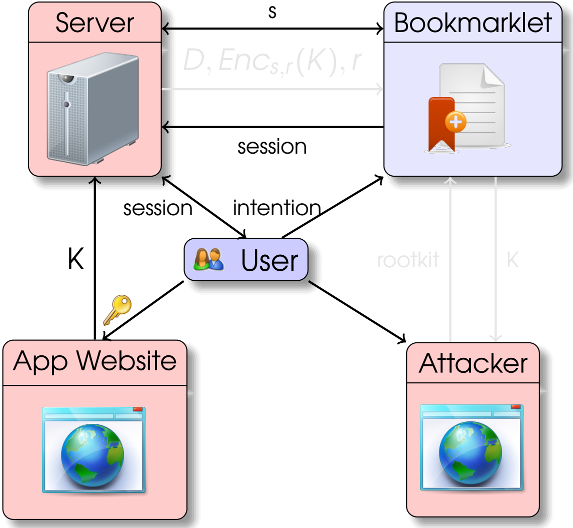
LastPass login bookmarklet



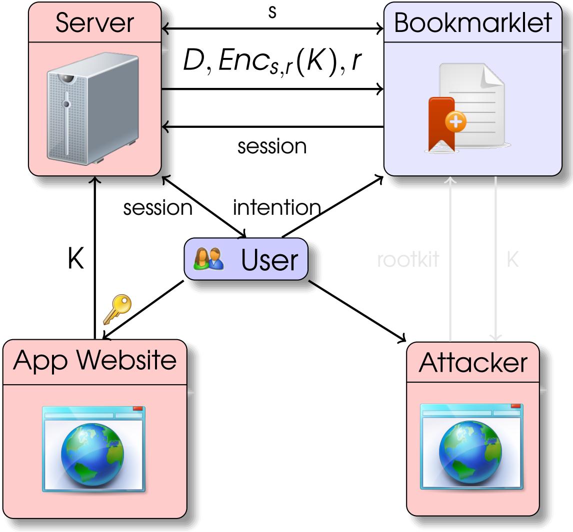
LastPass login bookmarklet



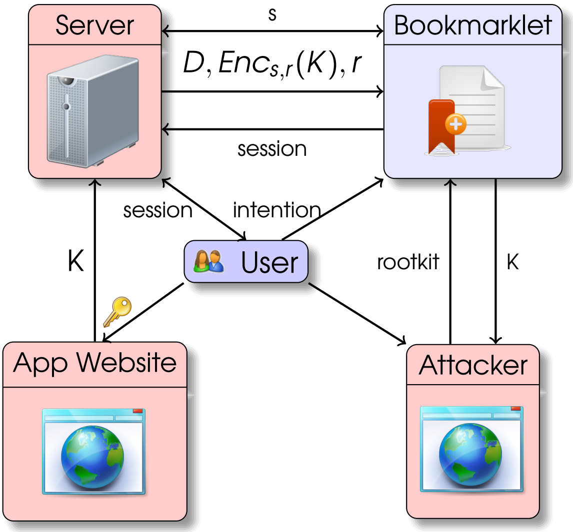
LastPass login bookmarklet



LastPass login bookmarklet



LastPass login bookmarklet



Key recovery by rootkiting

Key recovery by rootkiting

```
function _LP_START() {
 _LP = new _LP_CONTAINER();
 var d = {<encrypted form data>};
 _LP.setVars(d, '<user>',
 '<encrypted_key>', _LASTPASS_RAND, ...);
 _LP.bmMulti(null, null);
}
```

- Ben Adida, Adam Barth and Collin Jackson
  Rootkits for JavaScript environments
  WOOT'2009

Defensive JavaScript

Challenges of JavaScript static analysis

- Implicit initialization and global definition of undeclared variables.
- Dynamic property access and creation.
- Weak, dynamic types (`1+"x"`, `"1.1"==1.1`), implicit function calls for conversions (`valueOf`, `toString`).
- No distinction between functions, methods and constructors.
- No static scoping (`this`, `with`).
- Prototype chain inheritence, redefineable prototypes for base objects.
- Getters and setters.

Defensive JavaScript

Challenges of JavaScript static analysis

- Implicit initialization and global definition of undeclared variables.
- Dynamic property access and creation.
- Weak, dynamic types (`1+"x"`, `"1.1"==1.1`), implicit function calls for conversions (`valueOf`, `toString`).
- No distinction between functions, methods and constructors.
- No static scoping (`this`, `with`).
- Prototype chain inheritence, redefineable prototypes for base objects.
- Getters and setters.

Defensive JavaScript

Challenges of JavaScript static analysis

- Implicit initialization and global definition of undeclared variables.
- Dynamic property access and creation.
- Weak, dynamic types (`1+"x"`, `"1.1"==1.1`), implicit function calls for conversions (`valueOf`, `toString`).
- No distinction between functions, methods and constructors.
- No static scoping (`this`, `with`).
- Prototype chain inheritence, redefineable prototypes for base objects.
- Getters and setters.

Defensive JavaScript

Challenges of JavaScript static analysis

- Implicit initialization and global definition of undeclared variables.
- Dynamic property access and creation.
- Weak, dynamic types (`1+"x"`, `"1.1"==1.1`), implicit function calls for conversions (`valueOf`, `toString`).
- No distinction between functions, methods and constructors.
- No static scoping (`this`, `with`).
- Prototype chain inheritence, redefineable prototypes for base objects.
- Getters and setters.

Defensive JavaScript

Challenges of JavaScript static analysis

- Implicit initialization and global definition of undeclared variables.
- Dynamic property access and creation.
- Weak, dynamic types (`1+"x"`, `"1.1"==1.1`), implicit function calls for conversions (`valueOf`, `toString`).
- No distinction between functions, methods and constructors.
- No static scoping (`this`, `with`).
- Prototype chain inheritence, redefineable prototypes for base objects.
- Getters and setters.

Defensive JavaScript

Challenges of JavaScript static analysis

- Implicit initialization and global definition of undeclared variables.
- Dynamic property access and creation.
- Weak, dynamic types (`1+"x"`, `"1.1"==1.1`), implicit function calls for conversions (`valueOf`, `toString`).
- No distinction between functions, methods and constructors.
- No static scoping (`this`, `with`).
- Prototype chain inheritence, redefineable prototypes for base objects.
- Getters and setters.

Defensive JavaScript

Challenges of JavaScript static analysis

- Implicit initialization and global definition of undeclared variables.
- Dynamic property access and creation.
- Weak, dynamic types (`1+"x"`, `"1.1"==1.1`), implicit function calls for conversions (`valueOf`, `toString`).
- No distinction between functions, methods and constructors.
- No static scoping (`this`, `with`).
- Prototype chain inheritence, redefineable prototypes for base objects.
- Getters and setters.

Attacks to defend against

Scoping problem

Undeclared variables are implicitely global.

Attack example

```
function _LP_START() {
 _LP = new _LP_CONTAINER();
 var d = {<encrypted form data>};
 _LP.setVars(d, '<user>',
 '<encrypted_key>', _LASTPASS_RAND, ...);
 _LP.bmMulti(null, null);
}
```

Scoping problem

Solution

- We use a monomorphic type inference system.
- We forbid features that break lexical scoping: `arguments.caller`, `with(o)`
- We need to distinguish functions and methods.

Scoping problem

Solution

- We use a monomorphic type inference system.
- We forbid features that break lexical scoping: `arguments.caller`, `with(o)`
- We need to distinguish functions and methods.

Scoping problem

Solution

- We use a monomorphic type inference system.
- We forbid features that break lexical scoping: `arguments.caller`, `with(o)`
- We need to distinguish functions and methods.

Attacks to defend against

Implicit function calls

Some type casts implicitely call redefineable functions.

Attack example

```
// Attacker
Object.prototype.valueOf =
 function(){steal(this.secret)};
// Unsafe code
a = {secret:"x"} + 1
```

Implicit function calls

Solution

- Monomorphic operators.
- Exceptions for safe typecasts (logical negation).

Implicit function calls

Solution

- Monomorphic operators.
- Exceptions for safe typecasts (logical negation).

Attacks to defend against

Source code leaks

The source of functions published to the page is public.

Attack example

```
// Attacker
window.registerEventListener =
 function(t,f){steal(f+'')};
// Unsafe code
window.registerEventListener("message",
 function(m)
 {
  if(m=="secret") doAction();
 }
);
```

Source code leaks

Solution

Functions posted to the page must be wrapped in a function defined inside a `with` literal:

```
with({f:function(m){if(m=="secret") g();}})
 registerEventListener("message",
  function(m){return f(m);}
 );
```

Attacks to defend against

Prototype poisoning

Accessing or creating a non-literal property can cause calls to prototype functions.

Attack example

```
// Attacker
Object.prototype.__defineSetter__("secret",
 function(v){steal(v);}
);
// Unsafe code
var o = {};
o.secret = 123;
```

Prototype poisoning

Solution

- Completely literal definition of objects and arrays.
- No dynamic accessor (main restriction).
- Type inference infers minimal set of property that must be defined in object.
- When applied to literal object, verify object signatures are compatible.
- For arrays, check bounds on length.

Prototype poisoning

Solution

- Completely literal definition of objects and arrays.
- No dynamic accessor (main restriction).
- Type inference infers minimal set of property that must be defined in object.
- When applied to literal object, verify object signatures are compatible.
- For arrays, check bounds on length.

Prototype poisoning

Solution

- Completely literal definition of objects and arrays.
- No dynamic accessor (main restriction).
- Type inference infers minimal set of property that must be defined in object.
- When applied to literal object, verify object signatures are compatible.
- For arrays, check bounds on length.

Prototype poisoning

Solution

- Completely literal definition of objects and arrays.
- No dynamic accessor (main restriction).
- Type inference infers minimal set of property that must be defined in object.
- When applied to literal object, verify object signatures are compatible.
- For arrays, check bounds on length.

Prototype poisoning

Solution

- Completely literal definition of objects and arrays.
- No dynamic accessor (main restriction).
- Type inference infers minimal set of property that must be defined in object.
- When applied to literal object, verify object signatures are compatible.
- For arrays, check bounds on length.

Attacks to defend against

Functions and methods

A method used outside an object binds `this` to the global object.

Attack example

```
// Unsafe code
with({secret: "x",
  f:function(){this.secret = "y"}})
 (function(){ var g = f; g()})();
```

Functions and methods

Solution

- Two sets of rules for functions and methods (if `this` is used).
- Methods have an an additional condition: the object in which they are defined must have a signature compatible with the set of properties of `this` used in the function.
- Annoying special case for `with`-bound methods.

Functions and methods

Solution

- Two sets of rules for functions and methods (if `this` is used).
- Methods have an an additional condition: the object in which they are defined must have a signature compatible with the set of properties of `this` used in the function.
- Annoying special case for `with`-bound methods.

Functions and methods

Solution

- Two sets of rules for functions and methods (if `this` is used).
- Methods have an an additional condition: the object in which they are defined must have a signature compatible with the set of properties of `this` used in the function.
- Annoying special case for `with`-bound methods.

Type system

$$\langle\tau\rangle ::= \text{number} \mid \text{boolean} \mid \text{string} \mid \text{undefined}$$

$\mid \alpha, \beta$	Type variable
$\mid \tilde{\tau} \to \tau$	Arrow
$\mid \tilde{\tau}[\rho] \to \tau$	Method
$\mid [\tau]_n$	Final Array
$\mid [\tau]_{\geqslant k}$	Array schema
$\mid \rho*$	Final object
$\mid \rho$	Object schema

$$\langle\rho\rangle ::= \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

Scoping: function rule

$$\text{Fun}\ \frac{\begin{array}{c}\text{body} = (\text{var } y_1 = e_1, \dots y_m = e_m; s; \text{return } r) \\ \lambda = \text{fresh}() \qquad\qquad \tilde{\alpha} = \text{fresh}() \\ \forall j \leqslant m, \Gamma, f : \lambda, \tilde{x} : \tilde{\alpha}, (y_i : \mu_i)_{i<j} \vdash e_j : \mu_j \\ \Gamma, f : \lambda, \tilde{x} : \tilde{\alpha}, \tilde{y} : \tilde{\mu} \vdash s : \text{undefined}; r : \tau_r \\ \mathcal{U}(\lambda, \tilde{\alpha} \to \tau_r)\end{array}}{\Gamma \vdash \text{function } f(\tilde{x})\{\text{body}\} : \tilde{\alpha} \to \tau_r}$$

Object and Array accessors

$$\text{PropR}\ \frac{\tau = \text{fresh}() \qquad \Gamma \vdash e : \sigma \qquad \mathcal{U}(\{l : \tau\}, \sigma)}{\Gamma \vdash e.l : \tau}$$

$$\text{ArrR}\ \frac{\tau = \text{fresh}() \qquad \Gamma \vdash e : \sigma \qquad \mathcal{U}([\tau]_{\geqslant n+1}, \sigma)}{\Gamma \vdash e[n] : \tau}$$

Dynamic accessors

Adding dynamic checks

It's impossible to program without dynamic array accessors. We introduce a dynamic check that can be safely typed:

⟨*dyn_accessor*⟩ ::=
| (⟨*x*⟩ = @identifier) `[' ⟨*expression*⟩ `&' @posint `%' ⟨*x*⟩ `.length]'
| @identifier `[' ⟨*expression*⟩ `&' @posint `]'

$$\frac{\Gamma \vdash x : [\tau]_{\geqslant 1} \qquad \Gamma \vdash e : \text{int} \qquad n \in \mathbb{N}*}{\Gamma \vdash x[e\&n\%x.length] : \tau}$$

$$\frac{\Gamma \vdash x : [\tau]_{\geqslant n} \qquad \Gamma \vdash e : \text{int} \qquad n \equiv 0[2]}{\Gamma \vdash x[e\&n] : \tau}$$

Applications

Implementation

- We implemented a JavaScript parser and our type system in OCaml.
- We implemented defensive versions of HMAC-SHA-256 and AES-256-CBC and ensured that they were well-typed in our system.
- We used these primitives to build a safe version of the LastPass bookmarklet.

Applications

Implementation

- We implemented a JavaScript parser and our type system in OCaml.
- We implemented defensive versions of HMAC-SHA-256 and AES-256-CBC and ensured that they were well-typed in our system.
- We used these primitives to build a safe version of the LastPass bookmarklet.

Applications

Implementation

- We implemented a JavaScript parser and our type system in OCaml.
- We implemented defensive versions of HMAC-SHA-256 and AES-256-CBC and ensured that they were well-typed in our system.
- We used these primitives to build a safe version of the LastPass bookmarklet.

Perspectives

This work is incomplete

We are missing a formal security theorem about our type system.

Current problems

- Requires a formal semantics of JavaScript.
- Existing operational semantics by Sergio Maffeis lacks features that are critical to the security of our subset (getters and setters).
- Other alternatives (λJS, related IBEX results at Microsoft Research)?

Perspectives

This work is incomplete

We are missing a formal security theorem about our type system.

Current problems

- Requires a formal semantics of JavaScript.
- Existing operational semantics by Sergio Maffeis lacks features that are critical to the security of our subset (getters and setters).
- Other alternatives (λJS, related IBEX results at Microsoft Research)?

Perspectives

This work is incomplete

We are missing a formal security theorem about our type system.

Current problems

- Requires a formal semantics of JavaScript.
- Existing operational semantics by Sergio Maffeis lacks features that are critical to the security of our subset (getters and setters).
- Other alternatives (λJS, related IBEX results at Microsoft Research)?

Perspectives

Other extensions

- Automatic defensiveness transformation, automatic exploit generation.
- Subset extensions (constructors, dynamic memory allocation with computational security).
- New applications (single sign-on, client-side oauth)
- Translation of JavaScript into the WebSpi model in ProVerif.

Perspectives

Other extensions

- Automatic defensiveness transformation, automatic exploit generation.
- Subset extensions (constructors, dynamic memory allocation with computational security).
- New applications (single sign-on, client-side oauth)
- Translation of JavaScript into the WebSpi model in ProVerif.

Perspectives

Other extensions

- Automatic defensiveness transformation, automatic exploit generation.
- Subset extensions (constructors, dynamic memory allocation with computational security).
- New applications (single sign-on, client-side oauth)
- Translation of JavaScript into the WebSpi model in ProVerif.

Perspectives

Other extensions

- Automatic defensiveness transformation, automatic exploit generation.
- Subset extensions (constructors, dynamic memory allocation with computational security).
- New applications (single sign-on, client-side oauth)
- Translation of JavaScript into the WebSpi model in ProVerif.