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.

Goals

- Browser security
- Our contribution

Review of Host-Proof Web Applications

- Host-Proof Application Design
- Ciphertext Integrity
- URL Authentication
- Code/data separation
- Key management

Defensive JavaScript

- Attacks to defend against
- Type system
- Applications

Conclusion and Future Work
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

Introduction

Goals

Browser security
Our contribution

Review of Host-Proof Web Applications

Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Security Types for Web Applications
Antoine Delignat-Lavaud

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work

Web application overview
Web application overview
Introduction

Goals

Browser security

Our contribution

Review of Host-Proof Web Applications

Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript

Attacks to defend against
Type system
Applications

Conclusion and Future Work

Security Types for Web Applications
Antoine Delignat-Lavaud
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 believe 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 believe 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 believe 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 believe 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 believe is safe to use in such environments.
- We implemented a type system able to check if a given script belongs to that subset.
Introduction

Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Introduction
Goals
Browser security
Our contribution
Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management
Defensive JavaScript
Attacks to defend against
Type system
Applications
Conclusion and Future Work
Encrypted cloud storage

Server
- authentication
- encrypted data
- decryption script

User
- key
- decryption

App Website

Decrypted Data

Hacker
- sharing
- friends?

Introduction
- Goals
- Browser security
- Our contribution

Review of Host-Proof Web Applications
- Host-Proof Application Design
  - Ciphertext Integrity
  - URL Authentication
  - Code/data separation
  - Key management

Defensive JavaScript
- Attacks to defend against
  - Type system
  - Applications

Conclusion and Future Work
Encrypted cloud storage

**Server**
- authentication
- encrypted data
- decryption script

**User**
- key
- decryption

**App Website**
- decryption

**Hacker**
- CSRF
- XSS

**Decrypted Data**

- Sharing

**Introduction**
- Goals
  - Browser security
  - Our contribution

**Review of Host-Proof Web Applications**
- Host-Proof Application Design
  - Ciphertext Integrity
  - URL Authentication
  - Code/data separation
  - Key management

**Defensive JavaScript**
- Attacks to defend against
  - Type system
  - Applications

**Conclusion and Future Work**
Introduction

Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Security Types for Web Applications

Antoine Delignat-Lavaud

Introduction

Goals
- Browser security
- Our contribution

Review of Host-Proof Web Applications

Host-Proof Application Design
- Ciphertext Integrity
- URL Authentication
- Code/data separation
- Key management

Defensive JavaScript
- Attacks to defend against
  - Type system
  - Applications

Conclusion and Future Work

Encrypted cloud storage

Server

App Website

Hacker

User

Friends?

sharing

authentication

encrypted data

decryption script

Decrypted Data

decryption

key

Hacker

CSRF

XSS

Encrypted cloud storage
Password Manager Browser Extension

Server

App Code

User

Host Page

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Password Manager Browser Extension

Server

App Code

User

Host Page

Introduction
Goals
Browser security
Our contribution
Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management
Defensive JavaScript
Attacks to defend against
Type system
Applications
Conclusion and Future Work
Password Manager Browser Extension

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Password Manager Browser Extension

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Security Types for Web Applications
Antoine Delignat-Lavaud

Introduction
Goals
- Browser security
- Our contribution

Review of Host-Proof Web Applications
- Host-Proof Application Design
  - Ciphertext Integrity
  - URL Authentication
  - Code/data separation
  - Key management

Defensive JavaScript
- Attacks to defend against
  - Type system
  - Applications

Conclusion and Future Work
Introduction

Goals
- Browser security
- Our contribution

Review of Host-Proof Web Applications
- Host-Proof Application Design
  - Ciphertext Integrity
  - URL Authentication
  - Code/data separation
  - Key management

Defensive JavaScript
- Attacks to defend against
- Type system
- Applications

Conclusion and Future Work
Introduction

Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Introduction

Goals
- Browser security
- Our contribution

Review of Host-Proof Web Applications
- Host-Proof Application Design
- Ciphertext Integrity
- URL Authentication
- Code/data separation
- Key management

Defensive JavaScript
- Attacks to defend against
- Type system
- Applications

Conclusion and Future Work
Password Manager Bookmarklet

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Password Manager Bookmarklet

Introduction

Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Password Manager Bookmarklet

Introduction

Goals

Browser security

Our contribution

Review of Host-Proof Web Applications

Host-Proof Application Design

Ciphertext Integrity

URL Authentication

Code/data separation

Key management

Defensive JavaScript

Attacks to defend against

Type system

Applications

Conclusion and Future Work
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

Content Server

`google.com
ENC(u, p)`

User

Application Client

`ENC(u, p)`

Hacker

`google.com`
No ciphertext integrity protection

Introduction

Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
No ciphertext integrity protection

Friend

Content Server

User

Application Client

Hacker

- ENC(u, p)
- 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

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

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.

```
/^((?:([^:@]*)(?::([^:@]*))?)?@)?([^:/?#]*)((?:([^?#/@]*/)*)([^?#]*)?((?:[^@#]*)?))?$/
```

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
/^((?:([^:/?#]+):)?([^:/?#]+)?@)?([^:/?#]+)(?:([^/:]*))?([^/]*)(?:([^#]*))?([^#]*)?/\n
Incorrect
http://bad.com/#@accounts.google.com
```
Fishing attack on 1Password extension

URL parsing code

```javascript
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

```javascript
var href = getBrowser().contentWindow.
    .location.href + "/";

var domain = href.replace(
    /^http[s]*:\/\/(.*?)\/.*$/, "$1");

var middle = domain.replace(
    /^(www.)*(.*)/, "$2");

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

Fishing URL

http://www.google.com:xxx@bad.com
1Password fishing attack

Server

1Password

session

User

Attacker

Fishing URL

Google password

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof
Web Applications
Host-Proof Application Design
Ciphertext Integrity

URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and
Future Work
1Password fishing attack

Server

1Password

User

Attacker

session

Fishing URL

Google password

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity

URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
1Password fishing attack

- **Server**
- **1Password**
- **User**
- **Attacker**

**Session**

- Fishing URL
- Google password

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity

URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
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.
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.
Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Introduction

Goals
- Browser security
- Our contribution

Review of Host-Proof Web Applications
- Host-Proof Application Design
- Ciphertext Integrity
- URL Authentication

Code/data separation
- Key management

Defensive JavaScript
- Attacks to defend against
- Type system
- Applications

Conclusion and Future Work
Security Types for Web Applications
Antoine Delignat-Lavaud

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Query

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

Result

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

Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
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.
Security Types for Web Applications
Antoine Delignat-Lavaud

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Introduction

Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work

Security Types for Web Applications
Antoine Delignat-Lavaud

LastPass login bookmarklet

Server

Bookmarklet

App Website

Attacker

D, Enc_{s,F}(K), r

session

intention

rootkit

K

User

s
LastPass login bookmarklet

Server \[\text{session} \rightarrow D, \text{Enc}_{s,r}(K), r\] \[s\] Bookmarklet

User \[\text{intention} \rightarrow K\]

App Website

Attacker

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
LastPass login bookmarklet

Security Types for Web Applications
Antoine Delignat-Lavaud

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work

21 / 40
Introduction

Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Security Types for Web Applications

Antoine Delignat-Lavaud

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work

LastPass login bookmarklet

Server

Attacker

Bookmarklet

App Website

User

Attack

$D, Enc_{s,r}(K), r$

session

intention

rootkit

$K$

$K$

$s$
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 inheritance, 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 inheritance, 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 inheritance, 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 inheritance, redefineable prototypes for base objects.
  - Getters and setters.

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 inheritance, redefineable prototypes for base objects.
  - Getters and setters.
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 (\(\text{valueOf}, \text{toString}\)).
- No distinction between functions, methods and constructors.
- No static scoping (\(\text{this}, \text{with}\)).
- Prototype chain inheritance, 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 inheritance, 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 inheritance, redefineable prototypes for base objects.
- Getters and setters.
Scoping problem
Undeclared variables are implicitly global.

Attack example

```javascript
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 implicitly 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

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

Introduction
Goals
Browser security
Our contribution

Review of Host-Proof Web Applications
Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
Source code leaks

Solution

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

```javascript
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

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

- Two sets of rules for functions and methods (if `this` is used).
- Methods have 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 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 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

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

<table>
<thead>
<tr>
<th>\alpha, \beta</th>
<th>Type variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>\tilde{\tau} \rightarrow \tau</td>
<td>Arrow</td>
</tr>
<tr>
<td>\tilde{\tau}[\rho] \rightarrow \tau</td>
<td>Method</td>
</tr>
<tr>
<td>[\tau]_n</td>
<td>Final Array</td>
</tr>
<tr>
<td>[\tau] \geq k</td>
<td>Array schema</td>
</tr>
<tr>
<td>\rho^*</td>
<td>Final object</td>
</tr>
<tr>
<td>\rho</td>
<td>Object schema</td>
</tr>
</tbody>
</table>

\[ \langle \rho \rangle ::= \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} \]
Scoping: function rule

\[
\begin{align*}
\text{body} &= (\text{var } y_1 = e_1, \ldots y_m = e_m; s; \text{return } r) \\
\lambda &= \text{fresh()} \hspace{1cm} \tilde{\alpha} = \text{fresh()} \\
\forall j \leq m, \Gamma, f : \lambda, \tilde{x} : \tilde{\alpha}, (y_j : \mu_j)_{i < j} \vdash e_j : \mu_j \\
\Gamma, f : \lambda, \tilde{x} : \tilde{\alpha}, \tilde{y} : \tilde{\mu} \vdash s : \text{undefined}; r : \tau_r \\
\mathcal{U}(\lambda, \tilde{\alpha} \rightarrow \tau_r) \\
\Gamma \vdash \text{function } f(\tilde{x})\{\text{body}\} : \tilde{\alpha} \rightarrow \tau_r
\end{align*}
\]
Object and Array accessors

\[
\begin{align*}
\text{PropR} & \quad \tau = \text{fresh}() \\
\quad & \quad \Gamma \vdash e : \sigma \\
\quad & \quad U(\{l : \tau\}, \sigma) \\
\quad & \quad \Gamma \vdash e.l : \tau
\end{align*}
\]

\[
\begin{align*}
\text{ArrR} & \quad \tau = \text{fresh}() \\
\quad & \quad \Gamma \vdash e : \sigma \\
\quad & \quad U([\tau]_{\geq n+1}, \sigma) \\
\quad & \quad \Gamma \vdash e[n] : \tau
\end{align*}
\]
Dynamic accessors

Adding dynamic checks

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

\[
\langle \text{dyn\_accessor} \rangle ::= \\
\quad (\langle x \rangle = \texttt{@identifier}) \left( [.' \langle \text{expression} \rangle \&.' \texttt{@posint} \%.' \langle x \rangle \'.\text{length} ]' \\
\quad \texttt{@identifier} [.' \langle \text{expression} \rangle \&.' \texttt{@posint} '] \right)
\]

\[
\Gamma \vdash x : [\tau] \geq 1 \quad \Gamma \vdash e : \text{int} \quad n \in \mathbb{N}^* \\
\quad \Gamma \vdash x[e\&n\%x.length] : \tau
\]

\[
\Gamma \vdash x : [\tau] \geq n \quad \Gamma \vdash e : \text{int} \quad n \equiv 0[2] \\
\quad \Gamma \vdash x[e\&n] : \tau
\]
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.
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.
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.
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 ($\lambda$JS, 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 ($\lambda$JS, related IBEX results at Microsoft Research)?
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.

Introduction

Goals
- Browser security
- Our contribution

Review of Host-Proof Web Applications
- Host-Proof Application Design
- Ciphertext Integrity
- URL Authentication
- Code/data separation
- Key management

Defensive JavaScript
- Attacks to defend against
- Type system
- Applications

Conclusion and Future Work
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

Introduction

Goals
Browser security
Our contribution

Review of Host-Proof Web Applications

Host-Proof Application Design
Ciphertext Integrity
URL Authentication
Code/data separation
Key management

Defensive JavaScript
Attacks to defend against
Type system
Applications

Conclusion and Future Work
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.