Privilege separation in browser architectures

Enrico Steffinlongo

Università Ca' Foscari - Computer science

June 25, 2014

Browser Extensions

Web browsers extensions are phenomenally popular.

- roughly 33% of Firefox users have at least one add-on

Extension customize the user experience

- Customize the user interface
- Adds lots of functionality to the browser (e.g., save and restore tabs)
- Protect users from certain contents of the web pages

Browser Extensions

Extension need to interact with

- Web pages DOM
- Browser internal structure (tabs collections, ...)
- Browser API (browser storage, cookie jar, ...)

Potential security problem!

- Browser API $\Rightarrow$ security critical operations
- Web interaction $\Rightarrow$ Untrusted and potentially malicious

Chrome extensions architecture

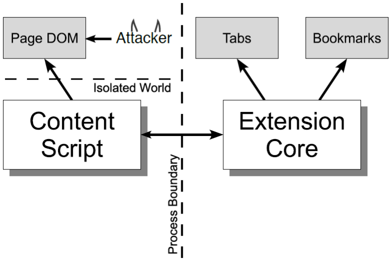
A chrome extension is composed by

- A manifest: the file containing all
- A set of Content scripts
- An Extension Core (composed by a set of scripts)
- Other resources

Chrome extensions architecture

Chrome extension architecture force developers to three practices

1. Privilege separation
2. Least privilege
3. Strong isolation



Privilege separation

- Content scripts
  - Injected to each page (multiple instances)
  - Access the DOM of the page
  - Cannot use privileges other than the one used to send messages to the Extension Core
- Extension Core
  - Single instance for each browser session
  - No access to DOM of pages
  - Can use privileges defined statically in the manifest

Least privilege

An extension has a limited set of permission defined statically in the manifest

- An extension cannot use more than required permissions
- User have to agree with the required permission at install time
- Attacker cannot use more than such set of privileges

Strong isolation

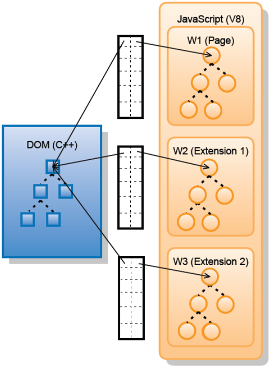
- Extension core is sandboxed in a process separated from the content scripts with unique origin
- Communication between Extension Core and Content Scripts is only via message passing
- Messages exchanged can only be string (Objects are marshaled using a JSON serializer without functions)
- Content script are executed in a isolated world from web pages

Isolated worlds

- Content script and web pages has different memory spaces
- Only standard DOM fields are shared

A potentially malign web page cannot:

- alter the content of variables of the content script
- invoke or share function with the content script



Message passing

to be fixed MPI

Chrome extension message passing API

Bundling

Extensions are often made by developer that are not security experts.

to be fixed Bundle

Developer tends to manage incoming messages in a centralized way. This is dangerous because

Example

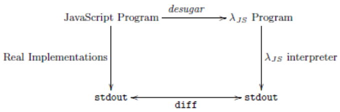
to be fixed Example

```
chrome.runtime.onMessage.addListener(
  function (msg, sender, sendResp) {
    if (msg.tag == "req") {
      var u = DB.getUser(msg.site);
      var p = DB.getPwd(msg.site);
      sendResp({"user": u, "pwd": p});
    }
    else if (msg.tag == "sync") {
      var db = DB.serialize();
      xmlhttp.open("GET", msg.site + db);
      xmlhttp.send();
    }
    else
      console.log ("Invalid message");
});
```

LambdaJS [1]

JavaScript:

- Complex language
- Lots of constructs
- unconventional semantics.



Very complex to analyze.

λ_{JS}[1] is a core calculus made by Brown university designed specifically to "desugar" JavaScript

- Few constructs
- Standard λ-style semantics
- Not a sound approximation of JavaScript
- Tests on "desugared" files shows that its the semantic coincide with JavaScript

Easy to analyze

The calculus

λ_{JS}++ is an extension of λ_{JS} with security oriented constructs. Its components are:

- Constants: $c ::= num \mid str \mid bool \mid \textbf{unit} \mid \textbf{undefined}$
- Values: $v ::= n \mid x \mid c \mid r_\ell \mid \lambda x.e \mid \{\overrightarrow{str_i : v_i}\}$
- Expressions:
$$\begin{array}{lcl} e & ::= & v \mid \textbf{let } x = e \textbf{ in } e \mid e\ e \mid op(\overrightarrow{e_i}) \mid \textbf{while } (e)\ \{\ e\ \} \\ & \mid & \textbf{if } (e)\ \{\ e\ \}\textbf{ else }\{\ e\ \} \mid e; e \mid e[e] \mid e[e] = e \\ & \mid & \textbf{delete } e[e] \mid \textbf{ref}_\ell\ e \mid \textbf{deref } e \mid e = e \\ & \mid & \overline{e}\langle e \triangleright \rho\rangle \mid \textbf{exercise}(\rho). \end{array}$$
- Memories: $\mu ::= \emptyset \mid \mu, r_\ell \overset{\rho}{\mapsto} v$
- Handlers: $h ::= \emptyset \mid h, a(x \triangleleft \rho : \rho').e$
- Instances: $i ::= \emptyset \mid i, a\{\!|e|\!\}_\rho$
- System: $s = \mu; h; i$

Judgments

For the analysis we used Flow logic [2]

Theorem

Let $s = \mu; h; \emptyset$. If $\mathcal{C} \Vdash s$ **despite** ρ, then s is $Leak_\rho(\mathcal{C})$-safe despite ρ.

Tool

We developed a tool in F# to perform the analysis described below. We:

1. add the chrome API definition as prelude to each source
2. desugar the source with prelude using the desugaring tool [1]
3. parse the desugared file using a YACC lexer/parser
4. alpha-rename all variables to avoid clashing since the analysis is context-insensitive
5. add annotation on the AST ($e \Rightarrow e^{\alpha}$)
6. generate the constraints for the AST
7. solve the constraints using a worklist algorithm
8. interpret the solution.

Compositional verbose

Constraint definition

Constraint generation

Worklist algorithm

Abstract domains

Results

Performance

Fst

a bunch of JavaScript code !_*(){}[]

adsad3

Future works

- Automatic correction of bundled extensions in order to debundle itself preserving its functionality
- Generalization of the analysis in order to check other similar architectures (e.g., Firefox)

Questions?

Thank you!

References

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi.
The essence of javascript.
In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag.

Hanne Riis Nielson and Flemming Nielson.
Flow logic: A multi-paradigmatic approach to static analysis.
In *The Essence of Computation*, pages 223–244, 2002.