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, . . .)

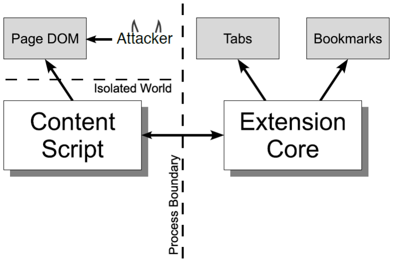
Potential security problem!

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

Chrome extensions architecture

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

All components of the extension have different address spaces except content scripts that can read and modify the DOM of the page on which are injected. So an infected component could not

- alter content of variables of other components
- invoke or alter functions defined in other components.

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)
- Functions cannot be sent.

Message passing

to be fixed MPI

Chrome extension message passing API

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 (Brown university[1])

JavaScript:

- Complex language
- Lots of constructs
- unconventional semantics.

Very complex to analyze!!

$$\begin{tikzcd}
\text{JavaScript Program} \arrow[r, "\textit{desugar}"] \arrow[d, "\text{Real Implementations}"'] & \lambda_{JS}\ \text{Program} \arrow[d, "\lambda_{JS}\ \text{interpreter}"] \\
\texttt{stdout} \arrow[r, leftrightarrow, "\texttt{diff}"'] & \texttt{stdout}
\end{tikzcd}$$

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

$\lambda_{JS}{++}$ is an extension of λ_{JS} with constructs specific for communications of components in privilege-separated architectures. 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: classical lambda calculus construct, operations on objects and on references, $e ::= \ldots \mid \textbf{ref}_\ell\ e \mid \overline{e}\langle e \triangleright \rho\rangle \mid \textbf{exercise}(\rho)$.
- 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 the Flow logic [2] approach.

- All values are abstracted assuming a pre-order relation $\sqsubseteq$.
  $v \rightsquigarrow \hat{v}$
- Abstract environment $\mathcal{C} = \hat{\Upsilon}; \hat{\Phi}; \hat{\Gamma}; \hat{\mu}$ is a four-tuple made of the following components:

$$\begin{array}{lcl} \textit{Abstract variable environment} & \hat{\Gamma} & : & \mathcal{V} \cup \Lambda \rightarrow \hat{V} \\ \textit{Abstract memory} & \hat{\mu} & : & \mathcal{L} \times \mathcal{P} \rightarrow \hat{V} \\ \textit{Abstract stack} & \hat{\Upsilon} & : & \mathcal{N} \times \mathcal{P} \rightarrow \mathcal{P} \times \mathcal{P} \\ \textit{Abstract network} & \hat{\Phi} & : & \mathcal{N} \times \mathcal{P} \rightarrow \hat{V}. \end{array}$$

We do not fix any specific representation of the domains.

Each judgment of our specification has the form:

- $\mathcal{C} \Vdash_{\rho} v \rightsquigarrow \hat{v}$
- $\mathcal{C} \Vdash_{\rho} e : \hat{v} \gg \rho'$
- $\mathcal{C} \Vdash \mu$ **despite** ρ, $\mathcal{C} \Vdash h$ **despite** ρ, $\mathcal{C} \Vdash i$ **despite** ρ, $\mathcal{C} \Vdash s$ **despite** ρ.

Example:

$$\text{(PE-C{\scriptsize OND})}\quad \frac{\begin{array}{c}\mathcal{C} \Vdash_{\rho_s} e_0 : \hat{v}_0 \gg \rho_0 \sqsubseteq \rho \\ \textbf{true} \in \hat{v}_0 \Rightarrow \mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \sqsubseteq \hat{v} \gg \rho_1 \sqsubseteq \rho \\ \textbf{false} \in \hat{v}_0 \Rightarrow \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \sqsubseteq \hat{v} \gg \rho_2 \sqsubseteq \rho\end{array}}{\mathcal{C} \Vdash_{\rho_s} \textbf{if } (e_0)\ \{\ e_1\ \}\ \textbf{else}\ \{\ e_2\ \} : \hat{v} \gg \rho}$$

Theorem

Permission leak against ρ : $\mathit{Leak}_\rho(\mathcal{C})$ is a sound over-approximation of the permissions which can be escalated by the opponent ρ in an initial system with arbitrary long call chains.

The main point of the analysis is this theorem:

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

This gives us statically an over-approximation of all permissions that an opponent can escalate to on a given extension.

Implementation steps

The analysis is implemented in a tool.

1. The analysis is turned in a Verbose approach (it saves all expressions values in a Cache),
2. The analysis of a program is turned in a finite set of constraints,
3. Is computed an acceptable solution starting from the set of constraints,

The translation from the Succinct approach to the Verbose one is made:

1. adding unambiguous labels $\alpha \in A$ to each expression: $e \Rightarrow e^{\alpha}$;
2. adding a Cache to the environment that stores all the partial result of each expression:

$$\textit{Abstract cache} \quad : \mathcal{L} \rightarrow \hat{V}$$

.

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 unambiguous labels α on nodes in 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

output

Worklist algorithm

output

Abstract domains

base

Performance

generate and solve preciso generate and solve sfigato generate and solve tricky generate and solve lazy

Results

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.