Privilege separation in browser architectures.

A static analysis approach.

Enrico Steffinlongo

Università Ca' Foscari - Computer science

April 9, 2015

Browser Extensions

Web browsers extensions are phenomenally popular.

- roughly 30% of Firefox users have at least one add-on

Extensions customize the user experience of the browser

- 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

Extensions need to interact with

- Web pages DOM
- Browser API (browser storage, cookie jar, . . .)

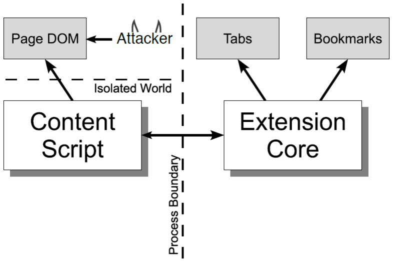
Potential security problem!

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

Chrome extensions architecture

Chrome extension architecture forces developers to adopt three practices

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



Privilege separation

- Content scripts
    - Are 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
    - Has a single instance for each browser session
    - Has no access to DOM of pages
    - Can use privileges defined statically in the manifest

Least privilege

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

- An extension cannot use more than required permissions
- Users have to agree with the required permissions 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 cannot

- 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 strings (Objects are marshaled using a JSON serializer)

Example

Content script 1:

```
chrome.runtime.sendMessage(
    {tag: "req", site: "www.google.com"});
```

Content script 2:

```
chrome.runtime.sendMessage(
    {tag: "sync",
    site: "www.password1.com"});
```

Attacker:

```
chrome.runtime.sendMessage(...);
chrome.runtime.sendMessage(
    {tag: "req", site: "www.google.com"});
chrome.runtime.sendMessage(
    {tag: "sync", site: "www.evil.com"});
```

Extension Core:

```
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[?])

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}[?] 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 semantic coincide with JavaScript

Easy to analyze!

The calculus (values and expressions)

$\lambda_{JS}{++}$ is an extension of λ_{JS}. It adds specific constructs for communications in privilege-separated architectures. Its components are:

- Constants: $c ::= \mathit{num} \mid \mathit{str} \mid \mathit{bool} \mid \textbf{unit} \mid \textbf{undefined}$
- Values: $v ::= n \mid x \mid c \mid r_\ell \mid \lambda x.e \mid \{\overrightarrow{\mathit{str}_i : v_i}\}$
- Expressions: classical lambda calculus construct, operations on objects and on references, send and exercise

$$\begin{array}{lcl} e & ::= & v \mid \textbf{let } x = e \textbf{ in } e \mid e\, e \mid op(\overrightarrow{e_i}) \mid \textbf{while } (e)\ \{\ e\ \} \\ & \mid & \textbf{if } (e)\ \{\ e\ \} \textbf{ else } \{\ e\ \} \mid e; e \mid e[e] \mid e[e] = e \\ & \mid & \textbf{delete } e[e] \mid \textbf{ref}_\ell\ e \mid \textbf{deref } e \mid \textbf{setref } e :\ e \\ & \mid & \overline{e}\langle e \triangleright \rho \rangle \mid \textbf{exercise}(\rho) \end{array}$$

Note that both $e[e] = e$ and **delete** $e[e]$ do not modify the object, but they returns a new updated object.

The calculus (memories, handlers, instances and system)

We added to the calculus architectural components in order to model the extension system.

- 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 [**?**] approach.

- All values are abstracted assuming a pre-order relation $\sqsubseteq$.
- We do not fix any specific representation of the abstract domains
- Abstract environment $\mathcal{C} = \hat{\Gamma}; \hat{\mu}; \hat{\Upsilon}; \hat{\Phi}$

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.

Judgment example

(PE-Cond)

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

(PE-App)

$$\frac{\begin{array}{c}\mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \\ \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \\ \forall \lambda x^{\rho_e} \in \hat{v}_1 .\ \hat{v}_2 \sqsubseteq \mathcal{C}_{\hat{\Gamma}}(x) \wedge \mathcal{C}_{\hat{\Gamma}}(\lambda x) \sqsubseteq \hat{v} \wedge \rho_e \sqsubseteq \rho\end{array}}{\mathcal{C} \Vdash_{\rho_s} e_1 \, e_2 : \hat{v} \gg \rho}$$

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:

$$\boxed{\begin{array}{l}\text{Let } s = \mu; h; \emptyset.\\ \text{If } \mathcal{C} \Vdash s \textbf{ despite } \rho\text{,then}\\ s \text{ is } \rho'\text{-safe despite } \rho \text{ for } \rho' = \mathit{Leak}_\rho(\mathcal{C}).\end{array}}$$

This gives us statically an over-approximation of all permissions that an opponent can escalate to if it attacks an extension.

Implementation steps

In order to develop a tool to perform such analysis

1. We turned the specification using a verbose approach that stores all expression values in a Cache,
2. The analysis of a program is turned in a finite set of constraints,
3. Starting from the set of constraints an acceptable solution is computed.

Compositional Verbose

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

1. unambiguous labels $\alpha \in A$ to each expression: $e \Rightarrow e^{\alpha}$;
2. a Cache to the environment that stores all the partial results of each expression: *Abstract cache* $\hat{C} : A \rightarrow \hat{V}$;
3. a Permission Cache to the environment that stores permissions used by each expression: *Abstract permission* $\hat{P} : A \rightarrow \rho$.

Compositional verbose

Example (if statement) in the verbose approach

$$
\begin{aligned}
[\mathit{CV\text{-}Cond}] \quad & \mathcal{CV} \Vdash_{cv,\rho_s} (\textbf{if } ({e_0}^{\alpha_0}) \ \{ \ {e_1}^{\alpha_1} \ \} \textbf{ else } \{ \ {e_2}^{\alpha_2} \ \})^{\alpha} \text{ iff} \\
& \mathcal{CV} \Vdash_{cv,\rho_s} {e_0}^{\alpha_0} \wedge \\
& \mathcal{CV}_{\hat{P}}(\alpha_0) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge \\
& \widehat{\textbf{true}} \in \mathcal{CV}_{\hat{C}}(\alpha_0) \Rightarrow \\
& \quad \mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV}_{\hat{C}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha) \wedge \\
& \quad \mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge \\
& \widehat{\textbf{false}} \in \mathcal{CV}_{\hat{C}}(\alpha_0) \Rightarrow \\
& \quad \mathcal{CV} \Vdash_{cv,\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{CV}_{\hat{C}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha) \wedge \\
& \quad \mathcal{CV}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha)
\end{aligned}
$$

Constraint definition

Constraints have this form:

$$\begin{array}{rrrcll} c & ::= & \{\hat{v}\} & \sqsubseteq & \mathsf{E} & \textit{Term inclusion} \\ & | & \mathsf{E} & \sqsubseteq & \mathsf{E} & \textit{Element inclusion} \\ & | & \widehat{Op}(\overrightarrow{\mathsf{E}_i}) & \sqsubseteq & \mathsf{E} & \textit{Operation inclusion} \\ & | & c & \Rightarrow & c & \textit{Implication} \end{array}$$

Where E is an index of the environment (e.g., $\Gamma(x)$).

Example: $\{\textbf{true}\} \sqsubseteq \mathsf{C}(\alpha)$ means that **true** is in the possible estimate of the node marked with α.

Constraints are generated from the program, and when they are created, they can be solved without the AST.

Abstract domains

As abstract domains for pre-values we used:

- All finite domains are abstracted in themselves,
- Signs $\oplus, \ominus, 0$ for numbers,
- Prefix for strings as described by Costantini, Ferrara, Cortesi in [**?**],
- Mapping from abstract strings to abstract values for objects.

An abstract value is a set of abstract pre-value.

Solving the constraints

Constraints can be solved together with the abstract domains definition using an existing fix-point constraint solver such as:

- ALFP succinct solver[**?**];
- BANE constraint solver (from Berkeley university);
- Z3 solver (from Microsoft research)
- worklist algorithm described in Principle of Programming Analysis [**?**].

We decided to use a custom implementation of the worklist algorithm.

Worklist algorithm

INPUT: $\mathcal{C}_{*\rho_s}[\![(e_*)^\alpha]\!]$
OUTPUT: (C, Γ, M)
METHOD:
Step 1: Initialization

```
W := [] : Queue(CElem)
D := [] : Map(CElem -> $\hat{V}$)
E := [] : Map(CElem -> Constraint)
for a in cache do
    Add (C a, $\perp$) D
    Add (C a, []) E
for x in vars do
    Add (R x, $\perp$) D
    Add (R x, []) E
for r in refs do
    Add (Mu ($\perp$, $\ell$), $\perp$) D
    Add (Mu ($\perp$, $\ell$), []) E
```

Step 2: Building the graph

```
for cc in lst do
    case cc of
    | {t} $\sqsubseteq$ p ->
      propagate p {t}
    | p1 $\sqsubseteq$ p2 ->
      Add cc E[p1]
    | {t} $\sqsubseteq$ p $\Rightarrow$ p1 $\sqsubseteq$ p2 ->
      Add cc E[p]
      Add cc E[p1]
    | $\widehat{op}(\overrightarrow{ps})$ $\sqsubseteq$ p1 ->
      for p in ps do
        Add cc E[p]
```

Worklist algorithm

Step 3: Iteration

```
while W $\neq$ [] do
    q = dequeue W
    for cc in E[q] do
        case cc of
        | p1 $\sqsubseteq$ p2 ->
            propagate p2 D[p1]
        | {t} $\sqsubseteq$ p $\Rightarrow$ p1 $\sqsubseteq$ p2 ->
            if t $\in$ D[p] then
                propagate p2 D[p1]
        | $\widehat{op}(\overrightarrow{ps})$ $\sqsubseteq$ p1 ->
            args = [D[p] | p $\in$ $\overrightarrow{ps}$]
            res = $\widehat{op}$ args
            propagate p1 res
```

Step 4: Recording the solution

```
for $\ell$ in $Ref_*$ do $\hat{\mu}(\ell)$ = D[Mu $\ell$]
for x in $Var_*$ do $\hat{\Gamma}(x)$ = D[R x]
for $\alpha$ in $Cache_*$ do $\hat{C}(\alpha)$ = D[C $\alpha$]
```

USING:

```
propagate q d =
    if d $\not\sqsubseteq$ D[q] then
        D[q] = D[q] $\sqcup$ d
        Enqueue q W
```

Performance

We developed various version of the worklist algorithm in order to increase the performances. We used

- a generalized version of the simple worklist algorithm,
- a more sophisticate implementation of the constraint generator that joins the implication constraint with same precondition,
- an optimized version of both constraints generator and solver based on lazy constraint generation as described in [**?**].

Benchmarks:

Number of nodes	Base	Optimized	Lazy
6268	8646.43 s	4171.89 s	0.94 s
9111	12974.89 s	6050.89 s	6.98 s
13075	29944.29 s	20334.26 s	12.21 s
20134	160845.38 s	29206.74 s	136.35 s

Tool

We developed a tool written in F$\#$ to perform the analysis. In the analysis we:

1. add the chrome API definition as prelude to each source
2. desugar the source with prelude using the desugaring tool [**?**]
3. parse the desugared file using lex/yacc
4. generate the constraints for the AST
5. solve the constraints using a worklist algorithm.

To increase precision of the analysis we also alpha-rename variables and inline some functions to avoid clashing since the analysis is context-insensitive.

Conclusion

In this work we developed:

- an analysis of a Chrome extension,
- a tool for checking real extensions.

The tool:

1. takes a real chrome extension,
2. analyzes it,
3. gives a static sound approximation of permissions that an opponent can escalate to.

Future works

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

Questions?

Thank you!

References