Chrome Extensions: Threat Analysis and Countermeasures

Lei Liu*
Vuclip Inc.
Milpitas, CA 95035
lliu@vuclip.com

Xinwen Zhang
Huawei R&D Center
Santa Clara, CA 95050
xinwen.zhang@huawei.com

Guanhua Yan
Los Alamos National Laboratory
Los Alamos, NM 87545
ghyan@lanl.gov

Songqing Chen
George Mason University
Fairfax, VA 22030
sqchen@cs.gmu.edu

Abstract

The widely popular browser extensions now become one of the most commonly used malware attack vectors. The Google Chrome browser, which implements the principles of least privileges and privilege separation by design, offers a strong security mechanism to protect malicious websites from damaging the whole browser system via extensions. In this study, we however reveal that Chrome's extension security model is not a panacea for all possible attacks with browser extensions. Through a series of practical bot-based attacks that can be performed even under typical settings, we demonstrate that malicious Chrome extensions pose serious threats, including both information dispersion and harvesting, to browsers. We further conduct an in-depth analysis of Chrome's extension security model, and conclude that its vulnerabilities are rooted from the violation of the principles of least privileges and privilege separation. Following these principles, we propose a set of countermeasures that enforce the policies of micro-privilege management and differentiating DOM elements. Using a prototype developed on the latest Chrome browser, we show that they can effectively mitigate the threats posed by malicious Chrome extensions with little effect on normal browsing experience.

1 Introduction

Web browsers such as Microsoft Internet Explorer (IE) and Mozilla Firefox are the main vehicle that people use to surf the Internet today. To enhance their functionalities and user experience, a large number of browser extensions have been developed by browser vendors or third party developers. The wide popularity of browser extensions has attracted the attention of attackers. A recent study [25] shows that nowadays many malware infections are in the form of add-ons on popular web browsers like IE and a majority of malware have a browser extension implementation. Browser helper objects (BHOs), a widely used type of add-ons, is named by CERT [24] as one of the techniques that are most frequently employed by spyware writers.

The threats posed by malicious browser extensions call for a thorough investigation of the security models that web browsers use to execute these extensions. Traditionally, as browser extensions run in the same process space as the browser itself, such as IE and Firefox, malicious web pages can exploit a buggy extension to steal users' sensitive data from the browser or other web pages that the browser is visiting. Moreover, the exploited extensions can even be used to attack the underlying operating system as they share the same privileges as the browser. To mitigate these threats raised under the traditional extension security model, a plethora of efforts have been made in the past few years, such as techniques to track behaviors in browser extensions [21, 25, 26], methods that monitor XPCOM calls made by extensions [29], and SABRE which tracks tainted JavaScript objects [20].

Meanwhile, a few new browser extension security models have been proposed as well [29, 20, 17, 19]. As a browser with security in design from scratch, the Google Chrome browser offers a built-in security protection mechanism for dealing with extensions. In Chrome, a typical extension consists of multiple components, including content scripts, an extension core, and optional native binary. Chrome runs each extension core in a separated process

*Work mainly performed at George Mason University.

from the browser process, and controls its access to browser objects and system resources with a permission profile [18]. The content script is injected into a tab when the web page is loaded, and runs in the same process space of the renderer of the web tab and can thus access its DOM objects. Injected content scripts in a tab can only communicate with the extension core via Chrome's inter-process communication channel (IPC). This design effectively enhances the security of the browser, as compromising an extension does not affect the execution of the browser. The Chrome extension support assumes that an extension would not launch active attacks against the browser or the underlying system; that is, all attacks are assumed to come from malicious web sites, e.g., those with malicious web pages or JavaScript code. By giving least privileges to an extension, Chrome expects that the attacking capability of a malicious web site through an extension is limited.

Although the multi-process architecture and the least privilege principle for extensions improve the overall security of the Chrome browser, it still lacks an effective protection mechanism against *malicious* extensions. While the Chrome extension security model mainly considers threats from malicious web sites, it does not protect the browser from being exploited by malicious extensions. For this extension security model to function correctly, users are required to install only trusted extensions. In practice, however, ensuring the trustworthiness of every browser extension is difficult, if not impossible. Google and the Chrome community encourage the development of extensions from third party developers. Currently, the Google Chrome extension gallery has about 10,000 extensions available for download from diverse developers. As suggested by previous analysis on Firefox add-ons [17], it is a daunting task to make sure that none of these extensions is malicious. On the other hand, there is no effective mechanism yet to prevent a user from installing extensions downloaded from other sources, e.g., embedded links from spam emails or phishing web pages. The recent Trojan posed as a fake Chrome extension suggests that such threats are not fictitious [16]. In fact, fraudulent and malicious Chrome extensions have been reported [7, 14], and Google is taking actions against malicious extension developers, e.g., one-time sign-up fee for an extension developer, and domain verification [5].

To investigate the current extension security model in Chrome, we conduct an experiment-based study with Chrome browsers. First, we demonstrate a few practical bot-based attacks with Chrome extensions. Through these extensions, we show that malicious attacks, such as information dispersion (e.g., email spam and DDoS), and information harvesting (e.g., password sniffing), can be easily mounted via cross-site HTTP requests and cross-site forgeries. Our in-depth analysis reveals that Chrome's incapability of preventing these attacks is rooted from the violation of the principles of least privileges and privilege separation. In particular, due to the assumption that all extensions are benign, Chrome unnecessarily offers the same set of permissions to the extension core and the content scripts simultaneously.

Following the principles of least privileges and privilege separation, we propose a set of countermeasures that enforce the policies of micro-privilege management and differentiating DOM elements. With the former, the extension core and the content script can have different types of permissions; with the latter, we can assign different sensitivity levels to the DOM elements of a web page. Our implementation works transparently with existing web applications and Chrome extensions. Using a prototype that works for the latest Chrome and older versions from version 7, we demonstrate that our new policies can effectively mitigate the threats posed by malicious Chrome extensions with little effect on normal browsing experience.

The rest of the paper is organized as follows. We describe the Chrome extension architecture and its security model in Section 2 and demonstrate various bot attacks in Section 3. We further perform detailed security analysis in Section 4 and propose a few countermeasures in Section 5, followed by the implementation in Section 6. We present the evaluation results in Section 7 and discuss some related work in Section 8. We make concluding remarks in Section 9.

2 Primer on Chrome Extensions

In this section we briefly describe some background about Chrome extensions, mainly about Chrome's extension architecture and the corresponding security model.

2.1 Architecture of Chrome Extensions

Chrome uses a multi-process architecture, where a single browser kernel process runs in the privileged mode to access platform and system resources, on behalf of multiple renderer processes. Each renderer process corresponds to a web page running as a tab. A renderer process runs in a sandboxed environment so it cannot directly access system resources such as the filesystem and the network. It can only send such requests to the browser process.

The design of Chrome extension architecture is based on the assumption that extensions are benign-but-buggy. The most possible attacks on Chrome extensions are malicious JavaScript from web pages. Thus the goal of the security architecture is to protect extensions from being exploited by these attacks, and control the potential damage done to the browser process if an extension is exploited. As an extension may access DOM objects in a web page and network resources, Chrome uses a multi-component architecture with fine-grained privileges to minimize its exposure to malicious web content.

In Chrome, an extension usually includes content scripts and an extension core. A content script is written in JavaScript that can be injected into a web page when the page is loaded. It then runs in the renderer process space to access the DOM tree. The extension core includes one or more background web pages written in HTML and JavaScript, and runs in a separate renderer process. The content script has the least privileges so it cannot access any object out of its renderer process space and has to communicate with the extension core via Chrome's inter-process communication (IPC). While the extension core contains the bulk of the extension privileges, it runs in a sandboxed environment. Therefore, it cannot access resources of the host platform and the network directly. It can only communicate with external web resources via `XMLHttpRequest`. Optionally, an extension can have binary code, which is launched as an NPAPI plugin. Chrome supports a `--safe-plugins` mode to run plugins in sandboxed environments. Note that in Chrome, plugins are different from extensions and are not part of the extensions we discuss here.

2.2 Security Model of Chrome Extensions

Chrome's extension architecture follows three security principles: *least privilege*, *privilege separation*, and *strong isolation*. Chrome defines a set of permissions, which includes executing native binary, accessing web sites, and accessing browser modules such as tabs, bookmarks, history, cookies, and geolocation. For the least privilege purpose, each extension has to declare the permissions that it requires in a `manifest.json` file, as part of the extension package. Chrome users are encouraged to download extensions from the Google-controlled extension gallery in order to avoid installing extensions with inappropriate privileges.

Chrome further separates privileges between different components of an extension. In particular, the content script of an extension can directly interact with web contents. However, by default it does not have the permissions to access browser modules, except that it can communicate to the extension core via `postMessage`. The extension core has most assigned privileges, but it is insulated from web pages. It has to use content scripts or invoke `XMLHttpRequest` to communicate with the web content. The native binary of an extension, running as an NPAPI plugin, has the most privileges as it can run arbitrary code or access any files.

To further control the damage of an exploited extension by malicious web pages, Chrome leverages three levels of strong isolations. First of all, privileges to access web sites are defined based on origins. Similar to other browsers, the basic security mechanism in Chrome is the isolation between origins, that is, the same origin policy (SOP) [13]. Chrome defines the origin of an extension by including a public key to the extension's URL, which is the origin of all its files in the local filesystem. By default, an extension can access all the resources of the origin. If an extension needs to access another origin, the cross-origin permission is required. Since the Chrome extension security model forbids an extension to access a web page directly by default, an extension has to access a web page by injecting some content scripts to the web page. To do this, a cross-origin request permission is also needed if the web page does not belong to the origin of the extension, which is often the case. If the permission is given, the injected content script can also access the origin of the web page by invoking `XMLHttpRequest`. Cross-origin permissions can be granted in `manifest.json` of the extension as follows.

```
"permissions":  [
   "tabs", "http://www.google.com/"
]
```

With the above permissions, the extension has the privilege to inject content scripts into a web page with the origin defined by URL `http://www.google.com`. Note that this permission also grants the extension core to access the specified origin (e.g., `http://www.google.com`) with the `XMLHttpRequest` method.[1]

The second level of isolation lies on the multi-process architecture of Chrome browser. Each component of an extension runs in different processes. The content script is injected into a tab and runs in the same process space of that sandboxed renderer process, while the extension core runs in another isolated process. The native binary of an extension runs in an isolated plugin process. This level of isolation helps preventing a compromised renderer process from accessing the extension core's functions.

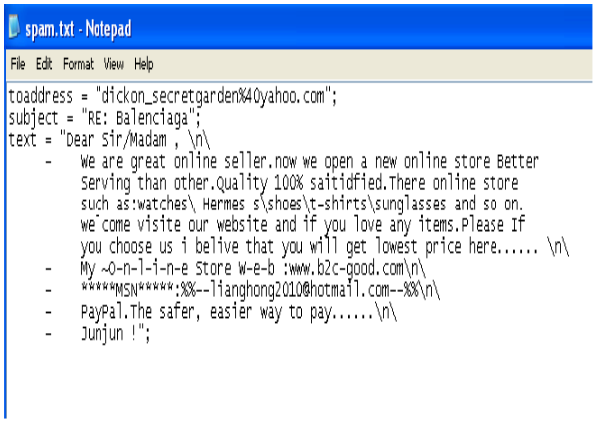
The third level of isolation is to run content scripts in a separate JavaScript engine, which is called `isolated world`. This provides an additional layer of isolation between the content script and the untrusted JavaScript environment of associate web pages. Content scripts from different origins run in different isolated worlds. Each isolated world has its own DOM copy, making it impossible to exchange JavaScript pointers.

Even with such a comprehensive security model, we find various attacks can still be mounted via Chrome extensions. We next will show how we exploit Chrome extensions for different attacks.

3 Attack Cases with Chrome Extensions

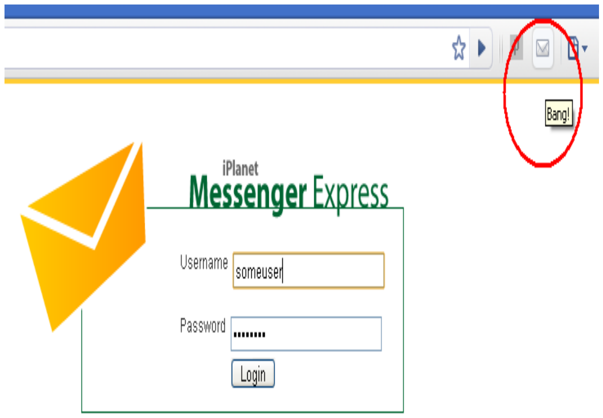
To demonstrate potential security risks of malicious Chrome extensions, we have developed an extension

[1] More permission description can be found at `http://code.google.com/chrome/extensions/manifest.html`.

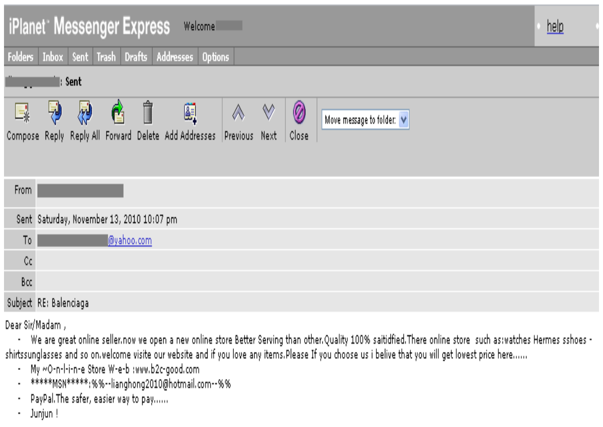


```
spam.txt - Notepad
File  Edit  Format  View  Help
toaddress = "dickon_secretgarden%40yahoo.com";
subject = "RE: Balenciaga";
text = "Dear Sir/Madam , \n\
    -   We are great online seller.now we open a new online store Better
        Serving than other.Quality 100% saitidfied.There online store
        such as:watches\ Hermes s\shoes\t-shirts\sunglasses and so on.
        we come visite our website and if you love any items.Please If
        you choose us i belive that you will get lowest price here...... \n\
    -   My ~O-n-l-i-n-e Store W-e-b :www.b2c-good.com\n\
    -   *****MSN*****:%%--lianghong2010@hotmail.com--%%\n\
    -   PayPal.The safer, easier way to pay......\n\
    -   Junjun !";
```

(a) Step 1: receiving spam content from the botmaster via extension update



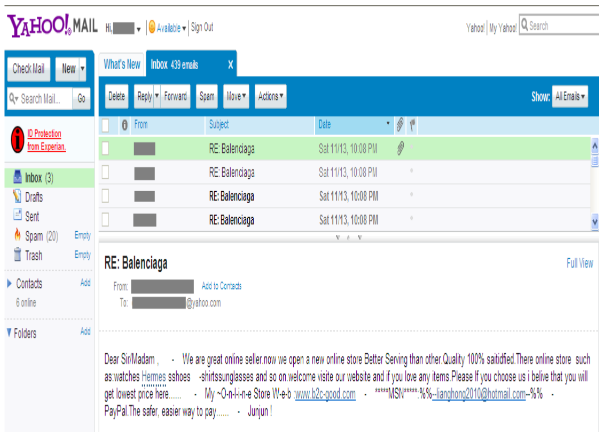
(b) Step 2: passively monitoring user's login with Bang! extension


iPlanet Messenger Express
Welcome [redacted]
help
Folders Inbox Sent Trash Drafts Addresses Options
[redacted]: Sent
Compose Reply Reply All Forward Delete Add Addresses Previous Next Close
Move message to folder:

From [redacted]
Sent Saturday, November 13, 2010 10:07 pm
To [redacted]@yahoo.com
Cc
Bcc
Subject RE: Balenciaga

Dear Sir/Madam ,
- We are great online seller.now we open a new online store Better Serving than other.Quality 100% saitidfied.There online store such as:watches Hermes sshoes -shirtssunglasses and so on.welcome visite our website and if you love any items.Please If you choose us i belive that you will get lowest price here......
- My ~O-n-l-i-n-e Store W-e-b :www.b2c-good.com
- *****MSN*****:%%--lianghong2010@hotmail.com--%%
- PayPal.The safer, easier way to pay......
- Junjun !

(c) Step 3: automatically sending out the spam email



From	Subject	Date
[redacted]	RE: Balenciaga	Sat 11/13, 10:08 PM
[redacted]	RE: Balenciaga	Sat 11/13, 10:08 PM
[redacted]	RE: Balenciaga	Sat 11/13, 10:08 PM
[redacted]	RE: Balenciaga	Sat 11/13, 10:08 PM

RE: Balenciaga
From: [redacted]
To: [redacted]@yahoo.com

Dear Sir/Madam , - We are great online seller.now we open a new online store Better Serving than other.Quality 100% saitidfied.There online store such as:watches Hermes sshoes -shirtssunglasses and so on.welcome visite our website and if you love any items.Please If you choose us i belive that you will get lowest price here...... - My ~O-n-l-i-n-e Store W-e-b :www.b2c-good.com - *****MSN*****:%%--lianghong2010@hotmail.com--%% - PayPal.The safer, easier way to pay...... - Junjun !

(d) Step 4: received spam email

Figure 1. Chrome Extension for Email Spam

Bang! that works for the latest Chrome and older versions from version 7. Bang! can be used as an automatic bot to mount large-scale email spamming, DDoS, and phishing attacks. Our attacks are based on the assumption that a malicious Chrome extension has already been installed on the Chrome browser.

For successful bot attacks, an essential component is a command and control channel. In our attacks, we leverage extension update for this purpose. Usually, a Chrome browser checks the update information of an extension from the update web site every few hours. If an update is available, the browser downloads the update and updates the extension files on disk. Therefore, this mechanism provides an ideal approach for establishing the command and control channel between the bot and the botmaster. The botmaster can simply use these periodic updates to distribute commands in the botnet, where the extension directly reads commands from an extension file without explicitly requiring any network access.

3.1 Email Spamming

Today botnets are notoriously responsible for most of spam emails on the Internet [11]. A spammer controls or rents a botnet and sends spamming commands to bots. After receiving spamming commands, bots send spam emails to victims. To defeat various malware detection mechanisms, a bot, instead of keeping sending spam emails at a high rate, can be instructed to send spam emails only sporadically, which makes detection more difficult.

To send out spam emails, Bang! has the following permission:

```
"permissions": [
   "tabs", "http://*/*", "https://*/*"
]
```

The `http://*/*` and `https://*/*` permissions are very common in popular extensions. They enable Bang! to send HTTP requests to all destinations. Table 1 shows the permissions of top 30 popular extensions (as of

Table 1. Permissions of Top 30 Popular Extensions

Rank	Name	Permissions
1	AdBlock	"http://*/*", "https://*/*", "contextMenus", "tabs"
2	Google Mail Checker	"tabs", "http://*.google.com/", "https://*.google.com/"
3	FastestChrome	"tabs", "http://*/*", "https://*/*"
4	IETab	"tabs", "bookmarks"
5	Browser Button for AdBlock	"http://*/*", "https://*/*", "tabs"
6	Docs PDF/PowerPoint Viewer	
7	Downloads	"tabs"
8	Google Translate	"http://*/*", "https://*/*", "tabs"
9	Facebook Photo Zoom	"contextMenus", "tabs", "http://*.facebook.com/*", "http://facebook.com/*", "https://*.facebook.com/*", "https://facebook.com/*"
10	Google Dictionary	"tabs", "http://*/*", "https://*/*"
11	Turn Off the Lights	"contextMenus", "tabs", "http://*/*", "https://*/*"
12	Firebug Lite	"tabs", "http://*/*", "https://*/*", "http://127.0.0.1/*", "http://localhost/*"
13	Download Master	"contextMenus", "cookies", "tabs", "http://*/*", "https://*/*"
14	Google Mail Checker Plus	"notifications", "tabs", "http://*/*", "https://*/*", "http://*.google.com/*", "https://*.google.com/*"
15	Adblock Plus	"tabs", "http://*/*", "https://*/*", "contextMenus"
16	RSS Subscription Extension	"tabs", "http://*/*", "https://*/*"
17	Clip to Evernote	"cookies", "tabs", "http://*/*", "https://*/*"
18	Google Chrome to Phone Extension	"contextMenus", "tabs", "http://*/*", "https://*/*"
19	Webpage Screenshot	"tabs", "http://*/*", "https://*/*"
20	Xmarks Bookmark Sync	"bookmarks", "tabs", "unlimited_storage", "http://*.xmarks.com/", "https://*.xmarks.com/", "http://*.foxmarks.com/", "https://*.foxmarks.com/", "http://*/*", "https://*/*"
21	SmileyCentral	
22	SocialPlus!	
23	Facebook for Google Chrome	"tabs", "http://*.facebook.com/"
24	Speed Dial	"bookmarks", "tabs", "http://*/*"
25	Google Voice	"tabs", "http://*.google.com/", "https://*.google.com/"
26	Cooliris	"tabs", "http://*/*", "https://*/*"
27	FlashBlock	"tabs", "http://*/*"
28	Smooth Gestures	"tabs", "bookmarks", "notifications", "idle", "cookies", "unlimitedStorage", "\u003Call_urls\u003E"
29	Awesome Screenshot	"tabs", "http://*/*", "https://*/*"
30	WOT	"tabs", "http://www.mywot.com/*", "http://api.mywot.com/*", "https://api.mywot.com/*"

May 18th 2011) from Chrome extension galley. As shown in the table, 19 extensions (out of 30) have been granted privileges of `http://*/*` or `https://*/*`.

We use the extension update for the command and control channel. Accordingly, the spam information is stored in a file called `spam.txt` under the extension directory, and the extension can read this file and then obtain spam information including victims' email addresses and spam content.

Figure 1 shows the email spam attack we have implemented via this extension. Basically, after acquiring the spam information that is shown in Figure 1(a), to send out spam emails, the extension still needs additional information such as an email account to login into a mail server. There are different ways to achieve the access to an email account. For example, this information can be saved in `spam.txt` which has been provided by the botmaster, or the spammer may have registered some free email accounts.

Our implementation does not need email account information beforehand. Instead, it utilizes the user's legitimate account when the user logins into her email system. This approach can evade detection more effectively, because spam emails are sent out when the user logins into her web email system as shown in Figure 1(b). In the figure, the extension `Bang!` is our bot extension to monitor the user login. In this example, we experimented with the popular iPlanet email system [15]. As the extension is granted the privilege of `"tabs"`, it listens to the update notification of tabs with the method of `chrome.tabs.onUpdated.addListener()`. When the user logs into a web email system, the login credential is represented by a session id (`sid`) and rewritten to the URL of subsequent HTTP requests. As the bot extension has the tab permission, it can listen to the tab update notice. With this credential information, an HTTP request to the iPlanet mail server is authorized to take any action on behalf of the user, instead of sending the user name and password in each transaction.

Figure 1(c) shows the sent email in the `sent box` of the user. In this example, the extension sends out the spam email through the mail server with the user's legitimate account.

Figure 1(d) shows the received spam email in the victim account. In this email spamming attack, our extension sends out the HTTP requests, which in turn triggers the web server to send spam emails to the victim. As the victim email address can be embedded in the extension (as in `spam.txt`), the bot can always obtain new victim emails by updating the extension from the botnet master's server, which is allowed by default in the Chrome ecosystem.

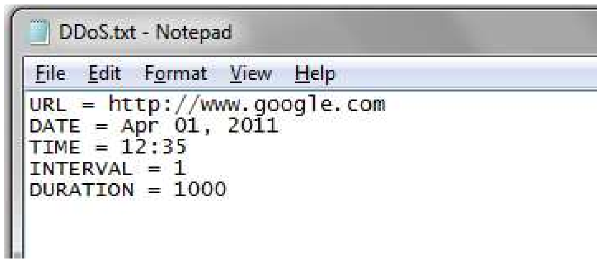
3.2 DDoS Attack

Although `http://*/*` is common, cautious users might prefer privileges with limited resource access, such as:

```
"permissions": [
   "tabs", "http://*.yahoo.com/*"
]
```

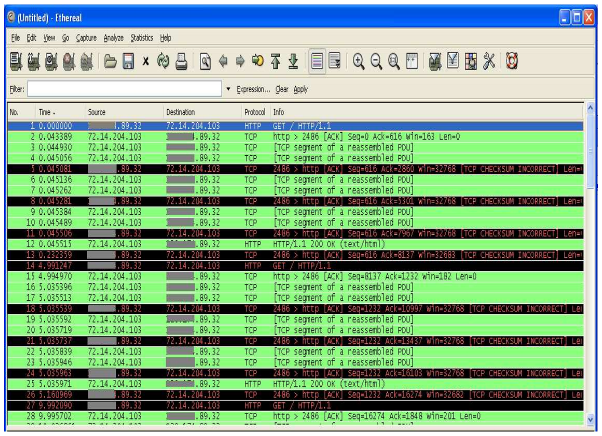
With above privilege, an extension can only access resources from `*.yahoo.com`, and should not be able to launch a DDoS attack against victims such as `www.google.com`. However, we note that the extension can inject content scripts into web pages from `*.yahoo.com`, and the injected content scripts have full privileges to access the DOM elements of the web pages. If the extension is malicious, it can change the `src` property of a DOM element to `www.google.com`, and therefore the injected content script can send out HTTP requests to the victim. Therefore, an extension does not need to make an explicit cross-site HTTP request to launch DDoS attacks. Figure 2 shows the DDoS attack against a victim server via this approach.

Figure 2(a) shows the command information that is obtained from an extension update. In this example, the DDoS information includes the victim's URL (`www.google.com`), attack start time (`12:35`), request interval (`1 second`), and attack duration (`1000 seconds`). After obtaining the victim's URL, the extension can make cross-site HTTP requests to the victim as



```
URL = http://www.google.com
DATE = Apr 01, 2011
TIME = 12:35
INTERVAL = 1
DURATION = 1000
```

(a) Step 1: receiving DDoS command from the botmaster via extension update


(Untitled) - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
No. Time      Source                 Destination            Protocol Info
  1 0.000000  [redacted].89.32       72.14.204.103          HTTP     GET / HTTP/1.1
  2 0.043389  72.14.204.103          [redacted].89.32       TCP      http > 2486 [ACK] Seq=0 Ack=616 Win=163 Len=0
  3 0.044930  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
  4 0.045056  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
  5 0.045081  [redacted].89.32       72.14.204.103          TCP      2486 > http [ACK] Seq=616 Ack=2860 Win=32768 [TCP CHECKSUM INCORRECT] Len=
  6 0.045136  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
  7 0.045262  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
  8 0.045281  [redacted].89.32       72.14.204.103          TCP      2486 > http [ACK] Seq=616 Ack=5301 Win=32768 [TCP CHECKSUM INCORRECT] Len=
  9 0.045384  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
 10 0.045489  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
 11 0.045506  [redacted].89.32       72.14.204.103          TCP      2486 > http [ACK] Seq=616 Ack=7967 Win=32768 [TCP CHECKSUM INCORRECT] Len=
 12 0.045515  72.14.204.103          [redacted].89.32       HTTP     HTTP/1.1 200 OK (text/html)
 13 0.232359  [redacted].89.32       72.14.204.103          TCP      2486 > http [ACK] Seq=616 Ack=8137 Win=32683 [TCP CHECKSUM INCORRECT] Len=
 14 4.991247  [redacted].89.32       72.14.204.103          HTTP     GET / HTTP/1.1
 15 4.994970  72.14.204.103          [redacted].89.32       TCP      http > 2486 [ACK] Seq=8137 Ack=1232 Win=182 Len=0
 16 5.035396  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
 17 5.035513  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
 18 5.035539  [redacted].89.32       72.14.204.103          TCP      2486 > http [ACK] Seq=1232 Ack=10997 Win=32768 [TCP CHECKSUM INCORRECT] Le
 19 5.035592  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
 20 5.035719  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
 21 5.035737  [redacted].89.32       72.14.204.103          TCP      2486 > http [ACK] Seq=1232 Ack=13437 Win=32768 [TCP CHECKSUM INCORRECT] Le
 22 5.035839  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
 23 5.035946  72.14.204.103          [redacted].89.32       TCP      [TCP segment of a reassembled PDU]
 24 5.035963  [redacted].89.32       72.14.204.103          TCP      2486 > http [ACK] Seq=1232 Ack=16103 Win=32768 [TCP CHECKSUM INCORRECT] Le
 25 5.035971  72.14.204.103          [redacted].89.32       HTTP     HTTP/1.1 200 OK (text/html)
 26 5.160969  [redacted].89.32       72.14.204.103          TCP      2486 > http [ACK] Seq=1232 Ack=16274 Win=32682 [TCP CHECKSUM INCORRECT] Le
 27 9.992090  [redacted].89.32       72.14.204.103          HTTP     GET / HTTP/1.1
 28 9.995702  72.14.204.103          [redacted].89.32       TCP      http > 2486 [ACK] Seq=16274 Ack=1848 Win=201 Len=0
```

(b) Step 2: sending DDoS packets

Figure 2. Chrome Extension for DDoS

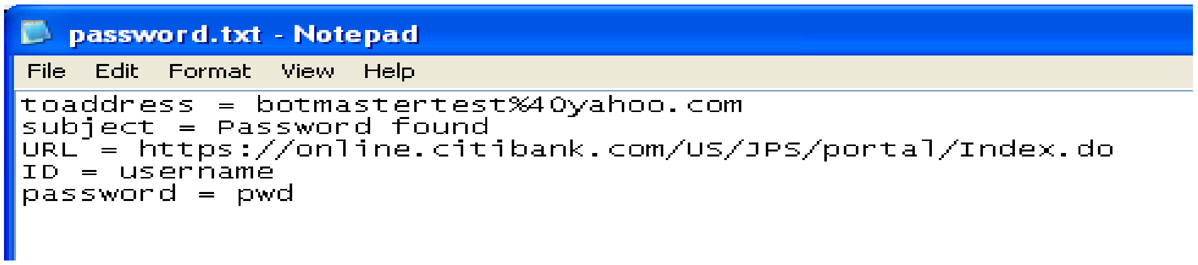
instructed by the DDoS command. Figure 2(b) shows the packet level traffic after the DDoS is initiated.

3.3 Password Sniffing

Nowadays, many Internet surfers use web browsers to do online shopping and access online bank accounts and financial services. Sensitive information such as bank account and password in these transactions is often saved by the web browser, temporarily or permanently, which makes web browsers a major target of spyware. We have implemented password sniffing in `Bang!` and in our experiment, the attack is against `online.citibank.com`. When the victim web page is loaded, `Bang!` injects content script into the web page, which can access all DOM elements including the form with the user name and password. Such information can then be sent to the designated email address.

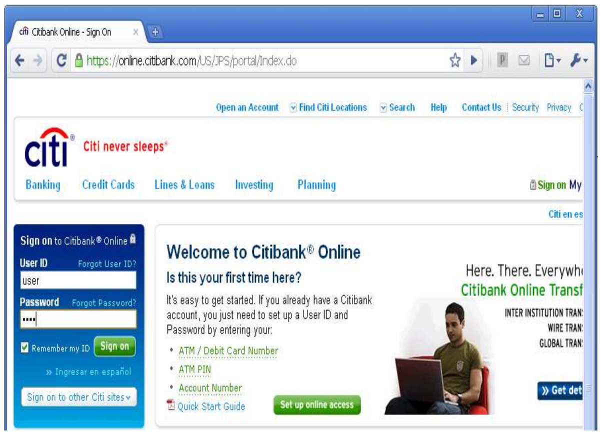
In order to access sensitive information in the Chrome browser, our extension needs to access the DOM tree of a web page. Therefore it needs the cross-site permission to insert the content script when a web page is rendered. The following manifest shows the permission specification.

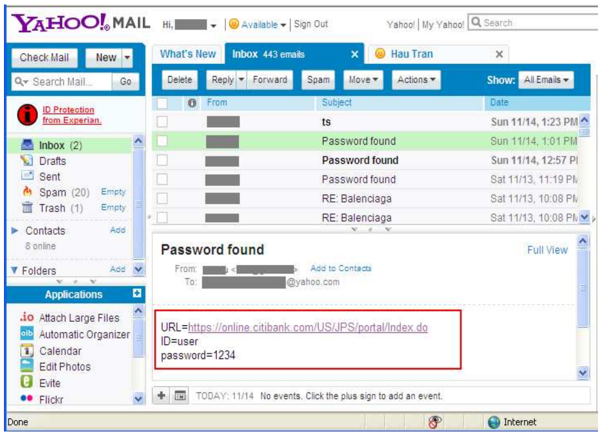
```
"content_scripts": [
```



```
password.txt - Notepad
File  Edit  Format  View  Help
toaddress = botmastertest%40yahoo.com
subject = Password found
URL = https://online.citibank.com/US/JPS/portal/Index.do
ID = username
password = pwd
```

(a) Step 1: receiving command from the botmaster via extension update





From	Subject	Date
[redacted]	ts	Sun 11/14, 1:23 PM
[redacted]	Password found	Sun 11/14, 1:01 PM
[redacted]	Password found	Sun 11/14, 12:57 PM
[redacted]	Password found	Sat 11/13, 11:19 PM
[redacted]	RE: Balenciaga	Sat 11/13, 10:08 PM
[redacted]	RE: Balenciaga	Sat 11/13, 10:08 PM

Password found
From: [redacted]
To: [redacted]@yahoo.com

URL=https://online.citibank.com/US/JPS/portal/Index.do
ID=user
password=1234

(b) Step 2: passively monitoring the user login

(c) Step 3: emailing the password to the botmaster

Figure 3. Chrome Extension for Password Sniffing

```
    {
    "matches":  ["https://online.citibank.com/*"],
    "js":  ["jquery.js", "myscript.js"]
     }
  ],
 "permissions":  [
    "tabs", "https://online.citibank.com/*"
  ],
    ...
```

With the above specification, when the user browses the page from `online.citibank.com`, two content scripts (`jquery.js` and `myscript.js`) are injected into the target web page, and the JavaScripts have full privileges to access all DOM elements including the form with user name and password. With the received command shown in Figure 3(a), `myscript.js` reads the values of user name and password elements when the user inputs, as shown in Figure 3(b), and sends to a designated email address. Figure 3(c) shows that the password information is successfully sent out with a similar mechanism as that in the previous DDoS attack. Note that with enough permissions such as cross-site access with `http://*/*` or `https://*/*`, the content script can also send sensitive information to the extension core, which in turn sends the data to the outside network.

4 Attack Analysis

We have demonstrated a few attacks with Chrome extensions in the last section. Given that a significant number of malware have been identified in the form of browser extensions in IE [25] and Firefox [27] and the number of Chrome users is on the rise [3], it is reasonable to believe that there will be more malicious Chrome extensions. In this section, we conduct a comprehensive analysis on the current Chrome extension security model to reveal its vulnerabilities that can be exploited by malicious extensions.

4.1 Threat Model

Similar to the design of Chrome extensions, in this study we assume the browser kernel and plugins are trustworthy. That is, we do not consider the attacks by exploiting the vulnerabilities in the browser and plugins. In addition, we assume that the underlying operating system, which provides sandboxing mechanisms for renderer processes and extension core processes, works benignly. Thus we do not consider attacks against the host system from malicious extensions. We further assume that a malicious extension does not use native code to launch attacks, e.g., all native code of an extension runs in a sandboxed environment. With the above exclusions, we focus on the attacks from malicious extensions against web applications including web pages rendered at the browser side and user data at the web server side.

In order to launch a successful attack, a malware instance typically needs two access authorizations: accessing sensitive information in the browser process and accessing the network for data transmission. The Chrome extension architecture makes these two accesses easily available to malicious extensions because they can not only access DOM and browser objects but also issue cross-site requests. By default, the content script of an extension has the permission to access all DOM and browser objects of a web page, including sensitive HTML elements such as password input, and modify the DOM tree, the page's URL, and cookie information. Furthermore, the content script can freely communicate with the origin of the associated web page via `XMLHttpRequest`. The remainder of this section shows that these default permissions pose serious threats to the browser with malicious extensions.

4.2 Cross-site Forgery with Content Script

An extension has to use some content script to access a web page's DOM objects. According to the current Chrome extension design, the content script cannot make cross-site requests without authorized permissions. In fact, a content script has the privilege of the origin of the associated web page, so it is capable of making HTTP requests to the current web page. Because the requests are regarded to have the same origin, all user credentials associated with the origin, such as cookies, can be included in the request. Many web sites use cookies as an authentication mechanism, which makes cross-site forgery attack possible when a content script abuses the trust between the browser and the server.[2] The email spam attack in the previous section could also be done with this approach.

To better understand this, consider another example in which an attacker lures the browser to load a well crafted image element `<img src="http://www.bank.com/withdraw?account=bob\&amount=1000000\&for=mallory"〉`. If the bank's web server keeps authentication information in a cookie and if the cookie has not expired, the attempt by the browser to load the image will submit the withdrawal form with the cookie, thus authorizing a transaction without the user's approval.

In practice, the above cross-site forgery attack may not be able to succeed, as usually a bank server only accepts a withdrawal transaction via an HTTP POST request rather than an HTTP GET request. By loading the image, the browser sends an HTTP GET request to `http://www.bank.com/withdraw`. If this is the case, the HTTP GET request is ignored. However, with malicious extensions, the attacker can launch a more sophisticated cross-site forgery attack by sending an HTTP POST request from the injected content script. Since the request includes user credentials such as cookies, the bank server can be fooled to authorize the transaction.

[2]We call this attack as cross-site forgery, as formally the content script has a different origin from the web page which it is injected into.

4.3 Cross-site Requests with Extension Core

Besides the cross-site forgery with content scripts, in a Chrome browser, cross-site requests can be made with the extension core as well. In Chrome, the content script of an extension can be injected into many web pages (tabs) concurrently in a browsing session, while the extension core of an extension runs as a global process that is able to communicate with content script instances in different tabs. This opens a door for a malicious content script to collude with the extension core to generate cross-site attacks.

By default, the content script injected into a web page cannot communicate with any other origin except the origin of the associated web page due to the same origin policy. With a malicious extension, however, its extension core can be used to make cross-site accesses to transfer sensitive data from one origin to another. In particular, suppose there are multiple tab processes concurrently running in a Chrome browser. Thus these tab processes share one extension core process. Assume tab 1 is a page from origin A and tab 2 is from origin B. As two tab processes are isolated, active web content (e.g., JavaScript embedded in the page) in tab 2 cannot access the web content in tab 1. However, with appropriate permissions, a malicious extension core can inject content scripts into tab 1 and 2 when the web pages are downloaded and rendered. The content script in tab 1 can access all information in the DOM of tab 1. Through IPC messages, the content script can forward the information to the extension core, which then passes the information to the content script injected into tab 2. With this approach, the content script in tab 2 can send information to the server of tab 2 via `XMLHttpRequest`, or write the information to the DOM of tab 2, which can be further read by embedded JavaScript of the web page in tab 2 and sent to the web server. Alternatively, the extension core can directly file cross-site HTTP requests to the origin of tab 2, due to the fact that the cross-site permission is enabled in order to allow the extension core to insert content scripts.

4.4 Unlimited Cross-site HTTP Requests

Without cross-site privileges, a running content script can only make HTTP requests to the same origin as the associated web page. However, the content script can access all DOM elements including modifying the DOM tree without requiring extra permissions. This capability enables a malicious content script to have unlimited cross-site HTTP requests. In particular, after being injected into a target web page, the content script can read any data content in the DOM tree, including sensitive data such as password. With this capability, the content script can insert an iframe element to the DOM tree, where the `src` property of the iframe element can be a malicious destination with the password, such as `http://evil.org/attack.msc?password=xxxx`. After this modification, the page is refreshed by the browser automatically, which in turn sends an HTTP GET request to the destination. The script in `attack.msc` at the `http://evil.org` can then handle this HTTP GET request and obtain the user password. Alternatively, the content script can also change the `src` property of an existing DOM element such as an image to trigger the HTTP request.

The root of the problem lies in the fact that the content script has full privileges to change the DOM of a page. As a result, it can add arbitrary new origins into DOM. To address this problem, a content script should request the privilege needed for introducing any new origin to a web page. By default, this capability should be denied by the browser.

4.5 Undifferentiated Permissions

Chrome permission specifications are based on origins. Once the privilege of an origin is granted, the extension can access almost all resources from that origin. These include the following: the extension can make cross-site requests to the origin with `XMLHttpRequest` and inject content scripts to the web page from this origin; the injected content script can access all DOM elements of the web page, and even introduce new origins to the web page by inserting or modifying the `src` property of an element. It is clear that once the privilege of one origin is granted, all extension components including the extension core and the content script have the same set of permissions to the origin. This is actually an all-or-nothing policy.

Consider an extension which usually needs to read contents from all web pages while only communicating to one particular web site, e.g., a language translator or a dictionary extension. The extension needs to inject content scripts to web pages in order to read the corresponding DOM content, which requires the cross-site permission to all the web origins specified as `http://*/*`. However, for a translator extension, the only reasonable and necessary cross-site request is to origins of the translator service web site. That is, in order to enable the content script injection, the extension is granted unnecessary privileges to many origins, rather than only the one it really needs. Obviously, the least privilege principle is not strictly enforced with this design. If we assume the extension is benign, the extra privileges may not hurt the browser. If used by a malicious extension, however, these extra privileges open a door for attacks, such as communicating command and control information for botnets and exfiltrating sensitive user data.

5 Security Enhanced Chrome Extensions

The previous section shows that the attacks demonstrated in Section 3 are made possible because malicious extensions are able to exfiltrate information through cross-site accesses and are granted unnecessary permissions for information stealing. Considering that the major security threat is either information dispersion or information harvesting attacks, we propose to use *micro-privilege management* that disables illegal cross-site accesses so as to prevent information exfiltration, and to *differentiate DOM elements with sensitivity* so as to prevent sensitive information harvesting.

5.1 Micro-Privilege Management

As shown in the attack examples, cross-site accesses result from the same privilege shared by different components of an extension. Such coarse-level privilege management, while convenient for some applications, leaves a loophole that extensions can exploit to achieve cross-site accesses. To enforce micro-privilege management, we propose to separate the privileges of different components first, and then assign the most appropriate privilege to each component.

Privilege Separation. Multiple components of a Chrome extension share the same set of access permissions for particular operations, which leads to non-least privilege for some components or operations. To address this problem, we need to strictly apply the privilege separation principle. In our approach, we not only separate the permission specifications for different components of an extension, but also separate the permissions of particular operations of the same component. Table 2 shows these two levels of permission separation. First, the privileges of the extension core and the content script are separated. Second, within each component, the privileges for different operations are specified by introducing new permission names. For example, new permissions, such as `inject_script` and `cross_site`, are defined to distinguish these two types of permissions.

Least Privilege. Security threats can come from extra permissions granted to extension components that are beyond necessity. To strictly follow the least privilege principle, we need to assign the most appropriate set of permissions (the least privilege) to each component. Thus, after separation, we further downgrade the default permissions of an extension and split existing permissions into fine-grained

Table 2. Micro-Privilege Management – Privilege Separation and Privilege Specification

	Permissions	Example Permission Spec
Extension core	`inject_script`	`"http://*/*",` `"https://*/*"`
	`cross_site`	`"http://www.translate.com"`
Content script	`sensitivity_level`	`"medium"`
	`same_origin_request`	`"false"`
	`new_origin`	`"http://www.translate.com"`

ones such that extra permissions can only be obtained via explicit requests.

Downgrading default permissions According to the analysis in the last section, a content script is allowed to access all DOM elements and modify the DOM tree without any permission check. Further, the content script is allowed to communicate with the origin of the associated web page freely. To prevent unlimited cross-site HTTP requests and sensitive information leakage, we should disable the capabilities of content scripts to introduce a new origin into a DOM tree and read sensitive information. To prevent cross-site forgery attacks, we should not allow a content script to send HTTP requests to the origin of the web page by default.

Fine-grained permission specifications Disabling default permissions certainly blocks many useful functions of benign extensions. To enable these functions whenever necessary for these applications, we define explicit permissions, including sensitive data accesses, requests to the origin of the associated web page, and introducing new origins to DOM. Without explicitly granted permissions, a content script cannot have the corresponding permissions. Table 2 summarizes fine-grained permission specification for content scripts.

With micro-privilege management, we can achieve better least privilege and privilege separation. Using a popular translation extension as an example, the following shows the corresponding permission manifest to mitigate the vulnerabilities identified in the previous sections.

```
"extension_core_permissions":  [
   "inject_script":[
       "http://*/*", "https://*/*"
    ]
   "cross_site":[
       "tabs", "http://www.translate.com"
    ]
 ]
"content_script_permissions":  [
   "sensitivity_level":[medium]
   "same_origin_request":[false]
   "new_origin":[
       "http://www.translate.com"
    ]
 ]
```

In the above specification, the permission that allows an extension core to inject content scripts is separated from that of cross-site HTTP requests. The translator extension has the privilege to inject content scripts to arbitrary web page `http://*/*/` while it only has the cross-site access privilege to a web site with origin `http://www.translate.com`. Even if the extension is malicious and can thus acquire sensitive information from the browser, it cannot send information to a malicious destination. Furthermore, the content script can only introduce origin `http://www.translate.com` into the DOM tree, which can be accessed by the extension by default. Therefore it cannot send any sensitive information to an arbitrary web site with cross-site HTTP requests by modifying the DOM tree. This effectively mitigates the vulnerability discussed in section 4.2.

Admittedly micro-privilege management on extensions is not a panacea for all possible attacks. For example, if the service origin itself (e.g., `http://www.translate.com`) is malicious, information leakage is still possible. Using our mechanism, a user or a system administrator however only needs to validate one origin rather than arbitrary origins, which significantly reduces the security risk.

5.2 Differentiating DOM Elements With Sensitivity

We have discussed that if we can control the access of sensitive information in DOM, we control the source of information harvesting attacks. The main challenge to do this lies in how to identify sensitive information in a web page.

To identify sensitive information in a web page, we can classify DOM elements based on their contents into three different sensitivity levels, from high to low. A straightforward approach to achieve this is to explicitly mark the sensitive information by web application developers or web site administrators (*however, for the usability and compatibility with existing extensions, we use our automatic tool Proctor for this purpose as presented in the next section*). An attribute of `sensitivity` can be assigned to a DOM element to represent the sensitivity level of information stored in the DOM element. If the `sensitivity` attribute is set, Chrome knows that the information content in this element is sensitive and will check the `sensitivity_level` permission of content scripts when they try to access the element value. For example, after an input element is marked with `high` sensitivity, when content scripts with a medium level permission attempt to access this input element, Chrome will forbid content scripts to acquire the value of the input element. More specifically, we introduce three levels of sensitivity as follows.

- *High level*: High level is defined to label the highly sensitive elements that are inherently sensitive. For example, an element of `<input type="password">` implies sensitive information in the `type`. Thus they should be protected with the highest priority. HTML has a list of such inherently sensitive elements noted by their types. Besides password, there are other types such as `hidden` that belong to this category.

- *Medium level*: Besides inherently sensitive elements, the attributes of a DOM element also hint their sensitivity. For example, a DOM element with the name *username*, highly likely, is related to a user name. They can be identified by their names, IDs or information format although the correlation is not always positive. Hence, we can build a dictionary that contains a number of regular expression patterns presenting sensitive information. When Chrome scans web contents and finds element names or IDs in the dictionary, the element should be marked with `Medium` level sensitivity.

- *Low level*: By default, all other elements are marked with low level sensitivity. Content scripts are free to access these elements.

An extension developer can assign one sensitivity level to content scripts at installation. Once the sensitivity level is granted by a user during installation, the content script obtains the permission and Chrome enforces the security check according to the protection level. The content script with high level sensitivity can access all elements in DOM. The medium level sensitivity allows content script to access elements with medium or low level sensitivity. By default, the content script with the low level sensitivity is forbidden to access elements with high or medium level sensitivity. If a content script with a medium level sensitivity attempts to read properties of a high level DOM element, Chrome will thwart this attempt. To be compatible with existing content script functions, Chrome does not simply return an error. Instead, a fake value is returned. For instance, when unauthorized content scripts read the value of a password input, Chrome will return a *** string with a random length. On the other hand, unauthorized write operations will be ignored.

6 Implementation

In this section, we discuss our prototype implemented, with both policies of micro-privilege management and differentiating DOM elements. In particular, to be compatible with existing extensions, we have implemented an extension for automatically labeling sensitivity levels of DOM elements as we shall discuss soon. Our implementation works with all Chrome versions from version 7.

To implement micro-privilege management, we have added finer permission definitions for each component of an extension. Existing Chrome only saves host permission. We use new variables to save the permissions for different components. Since content scripts have additional privileges, we define `sensitivity` levels, from high to low, and `same_origin_request` that represents whether content scripts are allowed to make same origin requests.

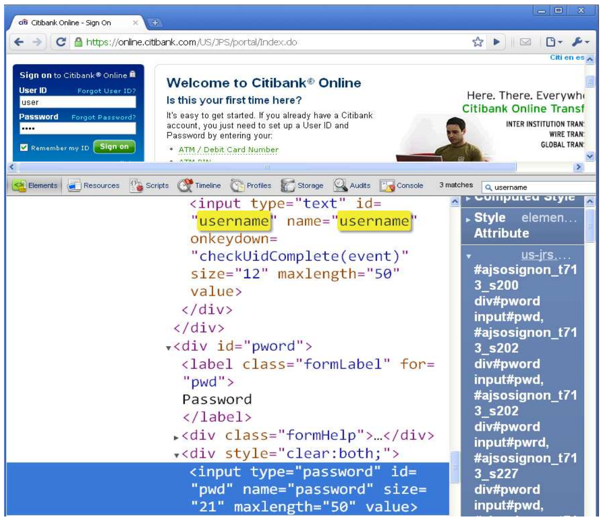
In previous sections, we indicate that one weakness of Chrome is that the content script can make illegal cross-site requests by modifying the `src` attribute of DOM elements. To thwart such illegal cross-site requests, we only allow content scripts to modify/append the `src` attribute to origins in the extension core's cross-site permissions because the extension core is capable of communicating with these origins as well.

Privileges are managed by the browser process and passed to render processes via struct `ViewMsg_ExecuteCode_Params`. We add new members to include finer privileges. To check fine-grained permissions on content scripts, we pass content script privileges to the JavaScript engine V8, where the permission information is saved.

To enforce security check for content scripts, we add security logic into a few methods. After obtaining the context of current JavaScript engine, Chrome knows whether it is in `Isolated World` or not. If it is not, the JavaScript must have come from web pages and no security check is needed. Otherwise, the sensitivity level of content scripts is checked and the `sensitivity` attribute of current element is accessed. For content scripts with lower sensitivity level requesting to read medium or high level DOM element value, the request is forbidden and Chrome returns fake value such as `"*****"` instead of the real value of the element.

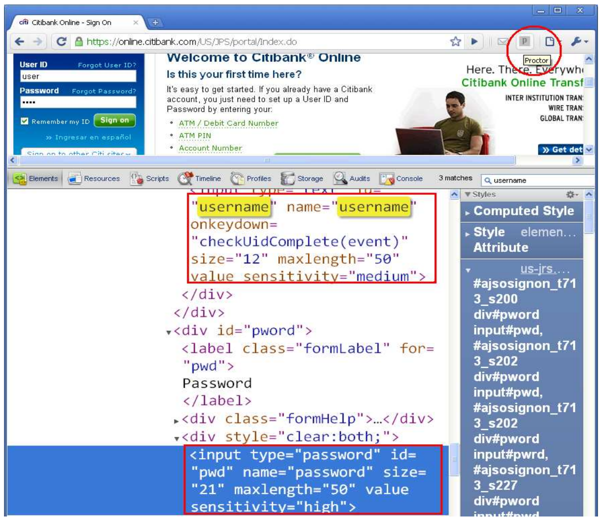
With a similar logic, security check is enforced when content scripts attempt to insert an element or modify the `src` attribute of an element. Our implementation guarantees that only origins with cross-site privilege are allowed.

To implement the policy of differentiating DOM elements, while it is best for the web application developer to denote distinct permissions for DOM elements, it may reduce the usability. To be compatible with existing web applications without bothering application developers, it is desirable to have a tool to mark the sensitivity levels of web contents upon loading automatically. For this, we implement a Chrome extension, called `Proctor`, which is a helper extension identifying and labeling sensitive elements. Once a DOM element is identified as high or medium sensitivity level, `Proctor` will set an attribute called *sensitivity* with the proper level to the element (e.g., `element.setAttribute('sensitivity', ''high''))`. Chrome itself is only responsible for querying the *sensitivity* attribute and enforces security protec-



```
<input type="text" id=
"username" name="username"
onkeydown=
"checkUidComplete(event)"
size="12" maxlength="50"
value>
</div>
</div>
▾<div id="pword">
<label class="formLabel" for=
"pwd">
Password
</label>
▸<div class="formHelp">…</div>
▾<div style="clear:both;">
<input type="password" id=
"pwd" name="password" size=
"21" maxlength="50" value>
```

(a) Step 1: Without Proctor Extension Installed



```
"username" name="username"
onkeydown=
"checkUidComplete(event)"
size="12" maxlength="50"
value sensitivity="medium">
</div>
</div>
▾<div id="pword">
<label class="formLabel" for=
"pwd">
Password
</label>
▸<div class="formHelp">…</div>
▾<div style="clear:both;">
<input type="password" id=
"pwd" name="password" size=
"21" maxlength="50" value
sensitivity="high">
```

(b) Step 2: After Proctor Extension Installed

Figure 4. Proctor Extension for Chrome

tion. The sensitivity dictionary can be flexibly updated. Currently, as a demonstration of concept implementation, `Proctor` separates sensitivity marking from Chrome. Ideally, its function should be integrated into Chrome in order to avoid any security concerns on the `Proctor` extension itself.

Figure 4 shows an example before and after the `Proctor` extension was installed. The sensitivity dictionary is the key to `Proctor`. According to HIPAA [8] and Chesapeake Research Review, Inc. [4], there are 18 types of individual identifiers from the security perspective including name, telephone number, social security number, account number, license number, etc. We also observed that DOM elements containing these individual identifiers usually have a similar name or ID. Accordingly, we define regular expression patterns in the dictionary for each sensitive identifier. By matching the element name or id to patterns in the dictionary, `Proctor` is capable of identifying potential sensitive elements.

To improve the accuracy of `Proctor`, besides element names and ids, `Proctor` matches the value of elements for further verification. Some sensitive information has a unique format. For example, an email address can be represented by `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b`. Thus, after `Proctor` matches element name or id, it matches the value of the elements to value patterns. If both matches are successful, `Proctor` has more confidence to mark the element as sensitive. Following the above idea, `Proctor` uses the following logic.

- If the elements already have a sensitivity attribute, `Proctor` respects it.
- If the type of the elements is password or hidden, `Proctor` marks its sensitivity as *High* level.
- If only the element name or its ID is matched, the element is marked with a *Medium* sensitivity level.
- After the element name or its ID has been matched, if the value of element is also matched, `Proctor` marks the element with a *High* sensitivity level.
- If neither is matched, Proctor ignores the element. By default, the element has a *Low* sensitivity level.

We show in Table 3 examples in our sensitivity dictionary we have currently implemented in extension `Proctor`.

Once a DOM element is identified as high or medium sensitivity level, `Proctor` will set an attribute called *sensitivity* with the proper level to the element (e.g., `element.setAttribute('sensitivity', "high")`.). Chrome itself is only responsible for querying the *sensitivity* attribute and enforces security protection. The benefit of `Proctor` is to separate sensitivity marking from Chrome. Proctor working as a Chrome extension also brings more flexibility with sensitivity dictionary update.

As shown in Figure 4, before `Proctor` is installed, there is not sensitivity attribute associated with the username and password elements. After `Proctor` is installed, when the web page is loaded, the username element is marked as *medium level* because the name of the element is matched from the dictionary (sensitivity attribute as medium) and the password element is marked as *high level* because the element has a password type (sensitivity attribute as high). After that, whenever content scripts attempt

Table 3. Dictionary of Proctor

```
"name_patterns": [
   {
   "social security number": ["SSN",
        "social security number"],
   "username": ["username", "uname"]
   "password": ["password", "pword", "pwd"]
   "email": ["email", "mail to"]
   "telephone": ["telephone", "tel"]
   "IBAN": ["IBAN", "bank account",
       "international bank account"]
       ...
    }
 ],
"value_patterns": [
   {
   "social security number": "^\d{3}-\d{2}-\d{4}$"]
   "! email":
       ["\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b"]
   "telephone":
       ["/^\(?(\d{3})\)?[- ]?(\d{3})[- ]?(\d{4})$/"]
   "IBAN": "[a-zA-Z]{2}[0-9]{2}[a-zA-Z0-9]
           [{4}[0-9]{7}([a-zA-Z0-9]?){0,16}"]
       ...
    }
 ],
   ...
```

to access elements with a sensitivity attribute, Chrome enforces access control on them.

7 Compatibility and Security Evaluation

To evaluate the effectiveness of our solution, we first study whether existing Chrome extensions in the Google Chrome Extension Gallery are compatible with the required modifications. As we target malicious extensions, we further evaluate whether the modified Chrome browser can prevent attacks via malicious extensions.

7.1 Compatibility Study

We first download the first 30 most popular extensions from the Google Chrome Extension Gallery and check their privilege configuration. Our analysis shows that among these 30 extensions, 24 have been granted network access permissions, among which 19 (about 80%) request higher privileges than necessary. Table 4 summarizes our analysis on these 30 most popular Chrome extensions.

For example, the most popular extension, `AdBlock` [1], requests the following privileges in mainfest.json:

```
"permissions": [
       "http://*/*", "https://*/*",
 "contextMenus", "tabs"],
   ...
```

With this permission set, `AdBlock` is able to make unlimited cross-site requests to all destinations although it does not need to make any cross-site requests at all. If it contains malicious code or is compromised, various attacks can be launched through it, as demonstrated in Section 3. Similar to `AdBlock`, most of the extensions from the Google Extension Gallery request permission “http://*/*/”, no matter whether they need the privilege or not. For example, among the top 30 popular Chrome extensions, besides AdBlock (No. 1), the following all have such a permission: Fastestchrome (No. 3), Browser Button for AdBlock (No. 5), Google Translate (No. 8), Google Dictionary (No. 10), Turn Off the Lights (No. 11), Firebug Lite (No. 12), Download Master (No. 13), Google Mail Checker Plus (No. 14), Adblock Plus (No. 15), RSS Subscription Extension (No. 16), Clip to Evernote (No. 17), Google Chrome to Phone Extension (No. 18), Webpage Screenshot (No. 19), Xmarks Bookmark Sync (No. 20), Speed Dial (No. 24), Cooliris (No. 26), FlashBlock (No. 27), Awesome Screenshot (No. 29).

On the other hand, many extensions also use content scripts to communicate with web contents. In many cases, this demands permissions `http://*/*/` and `https://*/*/`. However, these extensions should not make cross-site requests to arbitrary destinations. For example, `Auto-translate` [2] is an extension that automatically translates selected texts using `google translate`. As `Auto-translate` does not have its own translation service, it depends on the `google translate` service to conduct the translation. `Auto-translate` requires the following privileges:

```
"permissions": [
      "tabs", "http://ajax.googleapis.com/*",
      "http://*.google.com/*", "http://*/*",
      "http://google.com/*", "https://*/*"
 ],
   ...
```

Because `Auto-translate` uses content scripts to read selected texts in arbitrary web pages, it is reasonable to have the `http://*/*/` and `https://*/*/` privileges in order for content script injection. But it only requires cross-site requests to `http://translate.google.com`. Thus, it should not be granted the cross-site privilege to any destination other than `http://translate.google.com`. With our modified Chrome, the following permission will be given to `Auto-translate`:

```
"extension_core_permissions": [
   "inject_script":[
       "http://*/*", "https://*/*"
    ],
   "cross_site":[
       "tabs", "http://ajax.googleapis.com/*",
       "http://google.com/*",
       "http://*.google.com/*"
    ],
```

Table 4. Summary of Top 30 Chrome Extension Privilege Analysis

rank	name	over-privileged?	rank	name	over-privileged?
1	AdBlock	✔	16	RSS Subscription Extension	✔
2	Google Mail Checker	✗	17	Clip to Evernote	✔
3	FastestChrome	✔	18	Google Chrome to Phone Extension	✔
4	IE Tab	✗	19	Webpage Screenshot	✔
5	Browser Button for AdBlock	✔	20	Xmarks Bookmark Sync	✔
6	DocsPDF/PowerPoint Viewer	✗	21	SmileyCentral	✗
7	Downloads	✗	22	SocialPlus!	✗
8	Google Translate	✔	23	Facebook for Google Chrome	✗
9	Facebook Photo Zoom	✗	24	Speed Dial	✔
10	Google Dictionary	✔	25	Google Voice	✗
11	Turn Off the Lights	✔	26	Cooliris	✔
12	Firebug Lite	✔	27	FlashBlock	✔
13	Download Master	✔	28	Smooth Gestures	✗
14	Google Mail Checker Plus	✔	29	Awesome Screenshot	✔
15	Adblock Plus	✔	30	WOT	✗

```
 ],
"content_script_permissions": [
   "sensitivity_level":[low],
   "same_origin_request":[false]
 ],
   ...
```

With this configuration, `Auto-translate` can inject content scripts to read the texts, but it is only allowed to communicate with the `Google Translate` service. Our experiments with a number of websites show that `Auto-translate` performs translation without any problem. Note that under our configuration, even when `Auto-translate` contains malicious code or is compromised, its attack capability is still limited because it can only send sensitive information collected from the webpages it has accessed to `google.com`.

7.2 Bot Attack Mitigations

After compatibility study, we further test whether our prototype can defend against attacks we have shown before. For this purpose, we use `Bang!` to test the botnet attacks we have shown in Section 3.

In our modified Chrome browser, we assign the following permissions to `Bang!`.

```
"extension_core_permissions": [
   "inject_script":[
       "http://*/*", "https://*/*"
    ],
   "cross_site":[
       "tabs"
     ],
 ],
"content_script_permissions": [
   "sensitivity_level":[low]
   "same_origin_request":[false]
 ],
```

With the above specification, we repeat attacks on the security-enhanced Chrome with Proctor and Table 5 shows the results.

In the password sniffing attack, Figure 4 also shows the situation when the web page of citibank is loaded, before and after the `Proctor` extension is installed. If `Proctor` is installed on our modified Chrome browser, it marks the `password` input as `High` level sensitivity and `username` as `Medium` level sensitivity. When the malicious extension attempts to read the password, Chrome can detect that the extension only has a `Low` sensitivity privilege and thus returns a fake string "******" instead of the real password.

8 Related Work

Browser extensions have gained great popularity with add-on functions to enrich user browsing experience. However, due to insufficient security protection in the design and implementation of existing browsers, browser extensions today also pose significant threats to Internet users. Many extension vulnerabilities have been found and attacks have been reported in Firefox [27]. For untrusted browser extensions, some flow-tracking techniques have been developed to monitor their behaviors [21]. The extension and browser interactions have also been used for better protection [25, 26].

With increasing attacks by exploiting and compromising web browsers, lots of efforts have been made to re-define the browser architecture to enhance its security. These approaches include the OP browser [22], Gazelle [30], and IBOS [28]. All these browsers use a multi-process architecture to isolate different components of a browser, based on different isolation principles and granularity. However, these new architectures do not consider the threats from malicious browser extensions.

The Chrome browser [19] leverages a multi-component extension development to enforce the least privilege and privilege separation principles [18]. The strong isolation by

Table 5. Re-Evaluation of Bot Attacks

attack	result	reason
spamming	✗	unauthorized cross-site requests are completely blocked
DDoS	✗	unauthorized cross-site requests are completely blocked
password sniffing	✗	Proctor forbids sensitive information access
cross-site forgery	✗	content scripts are not allowed to make same origin request
unlimited cross-site requests by content scripts	✗	content scripts cannot change the `src` property of DOM elements to unauthorized origins

running different components in isolated processes provides privilege separation. Chrome, however, does not consider threats from malicious extensions, and therefore unnecessarily gives extra permissions to the extension core and the content script by default.

Mozilla Firefox [6] has a sandbox mechanism to provide indirect accesses through a wrapper to the DOM of a web page, and Mozilla runs an online extension gallery to recommend extensions that have been subjected to a review process using this sandbox technique. However, the sandbox mechanism relies on the discretionary compliance of web application developers [12]. In addition, recommended extensions are unfortunately still in the minority, in contrast to the large number of installed add-ons in Firefox [9]. Clearly, many developers do not submit Firefox extensions for reviewing.

JetPack [10] is a Firefox extension SDK. From the security perspective, it aims to reduce interaction interfaces between extensions and browser resources and functionalities. However, the current implementation of Jetpack technology is fully-privileged. That is, an extension developed with JetPack runs with the user's full privileges and has access to the complete Firefox extension API.

Ter-Louw et al. [29] were the first to address the security of JavaScript based extensions. However, as discussed before, their work was based on monitoring XPCOM calls; being coarse-grained, their approach leads to both false positives and negatives, and incurs a performance overhead of 19% for a particular policy. On the other hand, to prevent extensions from accessing sensitive information, SABRE keeps track of tainted JavaScript objects [20] and can deal with both exploited and malicious extensions. However, such an approach slows down all Javascript executions in the browser.

Besides aforementioned techniques with runtime permission restriction and containment, static analysis has also been proposed. Aiming to reduce human efforts in extension reviewing, VEX uses information-flow analysis on JavaScript code to identify potential security vulnerabilities in browser extensions [17]. Similar to Chrome, VEX does not detect malicious extensions. A Datalog-based policy language is recently proposed to specify and verify access control and data flow properties for browser extensions [23]. Targeting for extensions on variant browsers, this approach can specify very flexible and fine-grained security policies with DOM elements and other browser objects, and provides static analysis capability for extension developers and end users. As offline analysis tools, we believe these are complementary to real-time protection mechanisms such as the one proposed in this paper.

9 Conclusion

Recent years malware developers have increasingly exploited browser extensions for various attacks. In this study, we have conducted an experiment-based study on the security of the extension support in Google Chrome browsers. We have shown that under the existing security model for extensions in Chrome, it is not difficult to launch large-scale bot attacks. Through in-depth analysis, we find the problems are rooted from the coarse-grained privilege management for the extension components and undifferentiated access permissions for DOM elements in web pages. Accordingly, we propose new policies to enforce micro-privilege management and differentiate DOM elements, both of which have been implemented in our prototype. In particular, considering the compatibility with existing web applications, we develop an extension to automate the sensitivity assignment for different DOM elements. Our experiments show our design can effectively mitigate the security threats without affecting users' browsing experience.

Acknowledgment

We thank the anonymous referees for providing constructive comments. The work has been supported in part by U.S. AFOSR under grant FA9550-09-1-0071, and by U.S. National Science Foundation under grants CNS-0746649 and CNS-1117300 .

References

[1] Adblock, `https://chrome.google.com/extensions/detail/gighmmpiobklfepjocnamgkkbiglidom?hl=en-US`.

[2] Auto translate, `https://chrome.google.com/extensions/detail/obgoiaeapddkeekbocomnjlckbbfapmk`.

[3] Browser statistics, `http://www.w3schools.com/browsers/browsers_stats.asp`.

[4] Chesapeake irb, `http://chesapeakeirb.com/`.

[5] Chromium blog: Security improvements and registration updates for google chrome extensions gallery, `http://codeonfire.cthru.biz/?p=96`.

[6] Firefox web browser, `http://www.mozilla.com/en-US/firefox/firefox.html`.

[7] Google giving amazon top links in search results? no!, `http://www.seroundtable.com/google-amazon-treatment-13881.html`.

[8] Health information privacy, `http://www.hhs.gov/ocr/privacy/`.

[9] How many firefox users customized their browser? http://blog.mozilla.com/metrics/2009/08/11/how-many-firefox-users-customize-their-browser.

[10] Jetpack, `https://jetpack.mozillalabs.com/sdk/0.1/docs/#guide/security-roadmap`.

[11] Most spam comes from just six botnets, `http://en.wikipedia.org/wiki/Usage_share_of_web_browsers`.

[12] Mozilla sandbox review system, `https://addons.mozilla.org/en-US/firefox/pages/sandbox`.

[13] Same origin policy. `http://en.wikipedia.org/wiki/Same_origin_policy`.

[14] Smooth gestures spyware, `http://codeonfire.cthru.biz/?p=96`.

[15] Sun software product map,`http://www.oracle.com/us/sun/sun-products-map-075562.html`.

[16] Trojan poses as fake google chrome extension. http://www.bitdefender.com/NW1487-en–Trojan-Poses-as-Fake-Google-Chrome-Extension.html.

[17] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *Proc. of USENIX Security Symposium*, 2010.

[18] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecing browsers from extension vulnerabilities. In *Proc. of NDSS*, 2010.

[19] A. Barth, C. Jackson, C. Reis, and T. G. C. Team. The security architecture of the chromium browser. In *Stanford Technical Report, `http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf`*, 2008.

[20] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proc. of Annual Computer Security Applications Conference*, 2009.

[21] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of the 16th USENIX Security Symposium*, June 2007.

[22] C. Grier, S. Tang, and S. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[23] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *Proc. of IEEE Symposium on Security and Privacy*, 2011.

[24] A. Hackworth. Spyware. us-cert publication, 2005.

[25] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Proceedings of 15th USENIX Security Symposium*, August 2006.

[26] Z. Li, X. Wang, and J. Choi. Spyshield: Preserving privacy from spy add-ons. In *Proceedings of the 10th International Symposium, RAID*, 2007.

[27] R. S. Liverani and N. Freeman. Abusing firefox extensions. In *Defcon 17, `https://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-roberto_liverani-nick_freeman-abusing_firefox.pdf`*, 2009.

[28] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In *Proc. of the 2010 Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[29] M. Ter-Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal of Computer Virology*, (3), 2008.

[30] H. Wang, C. Grier, A. Moshchuk, S. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.