Defeat information leakage from browser extensions via data obfuscation

Wentao Chang, Songqing Chen

Department of Computer Science,
George Mason University,
Fairfax, VA 22030
U.S.A
{wchang7, sqchen}@gmu.edu

Abstract. Today web browsers have become the de facto platform for Internet users. This makes browsers the target of a lot of attacks. With the security considerations from the very beginning, Chrome offers more protection against exploits via benign-but-buggy extensions. However, more and more attacks have been launched via malicious extensions while there is no effective solution to defeat such malicious extensions. As user's sensitive information is often the target of such attacks, in this paper, we aim to proactively defeat information leakage with our iObfus framework. With iObfus, sensitive information is always classified and labeled automatically. Then sensitive information is obfuscated before any IO operation is conducted. In this way, the users' sensitive information is always protected even information leakage occurs. The obfuscated information is properly restored for legitimate browser transactions. A prototype has been implemented and iObfus works seamlessly with the Chromium 25. Evaluation against malicious extensions shows the effectiveness of iObfus, while it only introduces trivial overhead to benign extensions.

Keywords: Browser Extension, Chrome, Data Obfuscation, Information Leakage Threats.

1 Introduction

The web browser has become the de facto platform for everyday Internet users, and unarguably the driving force of the recent years' Internet revolution. Modern web browsers, such as Chrome, Firefox, and IE, are no longer simple static data renderers but complex networked operating systems that manage multiple facets of user online experiences[1][3].

To help browsers handle various emerging files and events, functionalities of web browsers are constantly enhanced. Most often such functionalities are extended by third-party code that customizes user experience and enables additional interactions among browser-level data and events. As of today, all major commodity web browsers support extensions. For example, Chrome has a list of over 10,000 extensions on Chrome Web Store by Dec 2010 [13].

However, the fact that web browsers have become the most popular vehicle for Internet surfing attracts more and more attacks. Among them, an increasingly popular attack is via browser extensions [7][8][10]. Commonly, these attacks are launched by exploiting existing extensions' weakness or by tricking user to install malicious extensions that can take over the web browser, steal cookies or user sensitive information without users' knowledge. For example, one of the earliest Firefox malicious extensions, FFsniFF [14] hides itself from the extension manager after it has been installed, monitors all form submissions in the browser for passwords and sends an email with gathered data to the attacker, and many Trojans disguise themselves as legitimate browser helper objects (BHO) for IE, but once installed they change user Internet settings and redirect users to random websites.

To deal with such threats, Google Chrome, one of the most popular web browsers, has made significant efforts by introducing several new security features to its extension framework [4][5]. It enforces strong isolation between web browsers and extensions, separates privileges among different components of extension, and uses a fine-grained permission system [11]. Recent studies [6][7] indicate that the Google Chrome extension framework is highly effective in preventing and mitigating security vulnerabilities in benign-but-buggy extensions that can be leveraged by web attackers. However, even Chrome does not cover all the bases, and most importantly it is defenseless to information dispersion or harvesting attacks launched by malicious extensions. For example, as these days online social networks are very popular, a rogue extension, Adobe business flash player [15], fetches and executes arbitrary JavaScript code from network once it has detected that the user has landed to certain social media websites. Users' social media accounts are then hijacked to post feeds or "like"s without users' consent. Other attacks can be launched to steal bank account information when users conduct online transactions as discussed in [7][12]. Existing work on extensions made little progress on the detection or protection of such attack vector.

To mitigate such an imperative threat, in this paper we design and implement iObfus. As users' sensitive information is the most critical asset, iObfus aims to defeat sensitive information leakage through browser extensions. It automatically classifies sensitive information on the web page with different default protection levels. Based on the protection policy, sensitive information will be automatically and passively obfuscated before any IO operations are performed. In this way, the users' sensitive information is always protected even information leakage happens (under this case only obfuscated information is leaked). To ensure the proper function of normal browser transactions, iObfus restores the context-aware sensitive information for legitimate transactions.

To demonstrate the effectiveness of iObfus, we build a proof-of-concept prototype on top of web browser Chromium 25. Experiments conducted against several malicious extensions in the wild show that iObfus can effectively protect user information from leaking. Further tests show that Chromium with iObfus does not interfere with normal daily transactions, and the data obfuscation and de-obfuscation cause trivial delay on users' experience.

The rest of the paper is organized as follows. Section 2 gives an analysis of the information leakage threats from Chrome extensions. We describe iObfus design and implementation in Section 3. An evaluation is conducted in Section 4. We discuss some related work in Section 5 and make concluding remarks in Section 6.

2 Information leakage threats from Chrome extensions

In this section, we discuss the information leakage threats from Chrome extensions. For the architecture and Chrome extension security model, please refer to Appendix A.

2.1 Threat analysis

Security concerns that online transactions could be hijacked or tampered with malicious extensions have arisen in recent years [7][12]. Password or financial information sniffing is one form of security attacks that malicious extensions could launch against web surfers. To access sensitive information such as the bank account number or login credentials, extensions need to inject Content Scripts to the victim web page. The injected Content Script will search in the DOM tree for elements of their interests, for example, <input> elements with type or name equal to “password” where user password is usually stored. To steal this information, attackers also need to establish a communication channel to the IP address where they hide. Thus malicious extensions also request cross-origin XHR permissions.

The recently popular attacks against Social Media accounts do not even need to steal users’ login credentials. Instead, the attackers try to masquerade as the users to engage social interactions stealthily. Such malware instances will check browser cookies to determine whether users have landed to certain social websites. If they have, another piece of JavaScript code will be fetched and executed, via which, the account can be used to spam your friends, post malicious links on news feed or follow/like other people or pages. This type of attacks seemingly acts like users’ normal behavior, thus they are unlikely to be detected by anti-virus program. Once infected, this threat tends to persist in user’s browser. As a matter of fact, this type of malicious extensions is the most common ones in the wild because attackers could gain monetary benefits and users are often not aware of the fact that they become victims.

The root causes of these attacks are: 1) Content Scripts have full access privileges to the DOM tree of the web page they are injected to, regardless of the fact that certain elements contain more sensitive information. If a fine-grained access control policy is enforced, we could control the source of information leakage. 2) The cross-origin XHR permission often grants access privileges to more origins than necessary. Each origin specified in the extension manifest file expands the target set of origins that the extension can leak information to. Since the extension core and the Content Script share the same set of origins, the potential sink points could scatter anywhere in Content Scripts or the extension core, making it difficult to track leakages.

2.2 Sources of information leakage through browser extensions

To defeat information leakage attacks, we need to first define the scope of “sensitive information” in the context of web surfing. The term “sensitive information” often differs in different research fields. In a broad sense, sensitive information includes but not limited to:

- Any information that can reveal users’ true identities or can be used to uniquely identify users, for instance, names, social security numbers, profile images, etc.
- Financial information or monetary equivalence such as bank account number, credit card number, digital currency, and so on.
- Any information from which others can infer users’ tendency or personal preferences, for example, users’ recent shopping list can indicate his/her lifestyle and be used for marketing purpose.

In general any data that users wish to withhold from others should be considered as sensitive and shall be protected cautiously by venders or service providers. The scope of sensitivity is so wide that in reality without a meaningful context of the term, there is little to be done to protect sensitive information practically. To defeat information leakage, it is essential to define the scope of sensitivity precisely in the context of browser extensions.

Based on our extensive study of possible leaking sources that are accessible to browser extensions, we classify sensitive information into two categories:

Per-tab user data from open web pages

When a user opens and views web pages, the multi-process Chrome browser will fetch each web page along with its resource files from the web server and render them in sandboxed render processes respectively. The extension core runs in its own process and does not have direct access to the memory space of sandboxed render process, however, Chrome allows extensions to inject Content Scripts to any web pages as long as the origin of the web pages match Content Scripts’ injection patterns. The injected Content Script has full access to the DOM tree of the targeted web page, thus sensitive information from per-tab user data becomes exposed to Content Script component of the extension.

Before Chrome 13, cross-origin XHR was not supported in Content Scripts, so that information leakage can only happen in the extension core. The message passing mechanism of Chrome extensions framework enables Content Scripts to send collected information back to the extension core. With proper message passing and receiving code implemented in the Content Script and the extension core, any user sensitive information on the web page is no longer local to its containing browser tab, and they are shared with the extension core, via which they can be further shared with other tabs.

We build our own set of privacy rules with the basis of HIPPA’s 18 rules [18] to identify candidates of sensitive per-tab data. The scope of set is dynamic depending on different websites; easy to expand/update and even let users choose their own tolerance level (The configurable level of sensitive information will be further studied).

Besides HIPPA rules, we also add to the scope DOM nodes with sensitive information specific to the browser, for example, sessions cookies, anti-forgery tokens that prevent Cross-Site Request Forgery (CSRF) attacks, etc.

Browser user data exposed by extension APIs

Modern browsers are allowed to maintain certain state information about their users aiming to remember user preferences and facilitate user actions. Such state information includes bookmarks of websites, download history, browsing history, cache of visited web pages, the chosen theme of browser UI, list of installed extensions, geolocation of browser, etc. The Chrome browser even offers its users to back up aggregated user settings via cloud services to their centralized Google account, so that the state information is synchronized across different Chrome instances.

The state information together is called browser user data and security measures should be taken by browsers to protect them from tampering and stealing by web and local attackers. In Chrome's approach, browser extensions are executed in a sandboxed environment and a permission system is used to regulate permissions assigned to extensions. If the principle of least privilege is strictly enforced, even they are taken over by attackers the damage to the browser should be contained.

However, Chrome's rich extension APIs and rough privilege definition make this situation complicated. In addition to all the APIs that web pages and Apps can use, Chrome also provides its own set of extension APIs to allow tight integration with the browser. While these APIs vastly enrich extension features, it also permits unfettered accesses to browser user data by extensions. Browser user data that are inherently safe in other browsers suffer from information leakage threats in the Chrome extension framework [26].

3 Design and implementation of iObfus

In this paper, we focus on protecting sensitive information that could be leaked through browser extensions. For this purpose, we design iObfus. We do not try to defeat information attacks launched by malicious websites in our system. We also assume malicious extensions discussed in this paper are written following the content security policy and are not easily detected by static analysis. Hence iObfus does not consider information leakage via *src* attributes of *img* tags, *iframe* tags, etc. Figure 1 sketches the architecture of iObfus and its working flow.

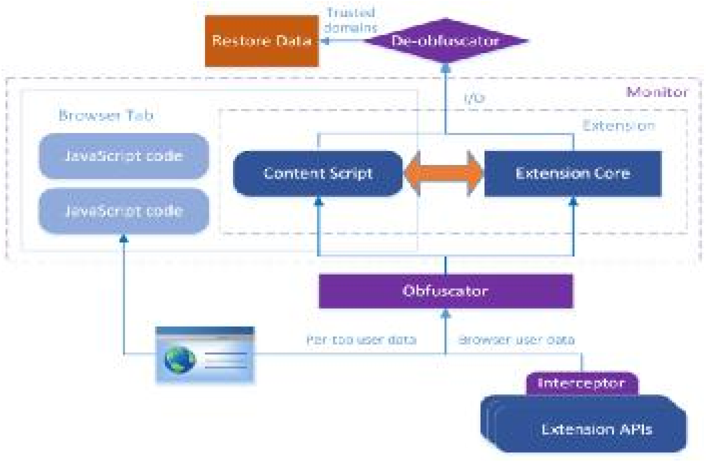


Fig. 1. iObfus architecture

We build the prototype on the open source Chromium project. Our prototype is compatible with all extensions developed for Chromium 25 and plus. What motivates us to build our prototype on Chromium is: (a) the Chrome browser is by far the most secure browser in the market and already has a comprehensive security model in place. (b) Building the prototype based on a platform that commercial browsers share the source code with also indicates that the security enhancement we propose can be easily transported to its commercial counterpart.

iObfus consists of four major components: Monitor, Interceptor, Obfuscator and De-obfuscator.

3.1 Monitor

This component monitors the execution of Content Scripts and JavaScript code in the extension core. Our system must be able to distinguish the execution of regular JavaScript code in web pages from JavaScript code introduced by extensions, which includes Content Scripts and JavaScript code running in background pages of the extension core. The ability to separate the execution of JavaScript code is important for two reasons. Firstly, limiting the scope of monitoring could reduce performance overhead of our system. Because iObfus aims to mitigate information leakage threats incurred by extensions in a cost effective manner, we can disregard attacks launched by JavaScript code of malicious websites and rely on the existing security model of Chrome extension framework to defeat them. Therefore, it is critical to identify origins of JavaScript code at runtime and enable/disable iObfus features on demand. Secondly, disabling obfuscation and restoration of sensitive per-tab data accessible to regular JavaScript code can avoid breaking the functionality of websites. For example, some normal behaviors in the web page such as the input validation of user login could be identified as a potential leakage and be obstructed by iObfus if we cannot exclude these JavaScript actions from our active monitoring.

We modify *compileAndRunScript()* method of *WebKit*'s *ScriptController* class to check whether the execution of JavaScript code snippet is within isolated world (please refer to Appendix A for definition). Only if it is, iObfus marks the separate copy of DOM documents as obfuscation candidates.

3.2 Interceptor

This component intercepts the subset extension APIs we identify that can expose browser user data to the extension core. As we discussed in the previous section, the browser user data is the second source of leakage, thus iObfus must be capable of instrumenting the subset extension APIs and obfuscating the browser user data before they are read by the extension core. Since the Chrome extension APIs are under active development, it is common that more experimental APIs become supported APIs in future releases. We perform the identification of leaking APIs from all extension APIs currently available in Chromium 25, and the list of APIs that iObfus intercepts is shown in Appendix B.

3.3 Obfuscator

The goal of Obfuscator is to protect user sensitive information without breaking the normal functionalities of extensions. The values of those potential leakage sources are obfuscated before they enter the memory space of extensions. Extensions can still access those data objects and use their values for actions as if they are regular JavaScript objects.

iObfus only obfuscates sensitive information derived from DOM documents that have a marked sensitivity flag by the Monitor or browser user data from intercepted extension APIs by the Interceptor. According to HIPAA and Chesapeake Research Review, Inc. [19], from the security perspective there are 18 types of individual identifiers including name, telephone number, social security number, account number, etc. Based on our observation and previous research [7], DOM elements containing sensitive information often have name, type or ID attributes with values correlating to these individual identifiers. For example, the HTML *input* element for password in Google account sign-in page has attributes of *type="password"*, *name="Passwd"* and *id="Passwd"*. Hence, a set of regular expression patterns is defined in iObfus to locate the first source of leakage – sensitive per-tab user data.

Due to the fact that the regular JavaScript code and the Content Script are executed intermittently, to avoid repetitive processing, iObfus also assigns an *"isProcessed"* Boolean flag to candidate DOM documents marked by the Monitor. Only if the value of *"isProcessed"* is false, iObfus begins to iterate every element of the DOM tree to find a pattern match for names, types and IDs. For matched DOM elements, iObfus processes them differently depending on their element type. For text node, iObfus applies the obfuscation algorithm to convert the text content into its obfuscated form and then call *replaceEntireText()* method to substitute the entire content of text node;

for element node, iObfus only replaces the content of *"value"* attribute with obfuscated data.

Table 1. Regular expression of some common data formats

SSN #	\d{3}-\d{2}-\d{4}$
Email Address	\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b
Url	^(http|https|ftp)\://([a-zA-Z0-9\.\-]+(\:[a-zA-Z0-9\.&amp;%\$\-]+)*@)*((25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9])\.(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[0-9])|localhost|([a-zA-Z0-9\-]+\.)*[a-zA-Z0-9\-]+\.(com|edu|gov|int|mil|net|org|biz|arpa|info|name|pro|aero|coop|museum|[a-zA-Z]{2}))(\:[0-9]+)*(/($\|[a-zA-Z0-9\.\,\?\'\\\+&amp;%\$#\=~_\-]+))*$

A context-aware obfuscation algorithm is the key to the success of the Obfuscator. The "obfuscated" form has to be syntactically equivalent to its original form so that the evaluation of these DOM objects at runtime does not fail. In the current prototype, iObfus defines regular expressions for some most common data formats as known contexts, and they are listed in Table 1. Before applying a randomization-based generic obfuscation, iObfus examines the input for known contexts. If a pattern match is detected, it will instead perform a context-aware transformation. For example, if iObfus detects an email address smith@gmail.com in its input, it will transform the email address into a fake one such as hnrgs@ymail.com, which can be restored later for legitimate I/O operations.

3.4 De-obfuscator

The De-obfuscator is responsible for URL and payload inspection of cross-origin XMLHTTPRequests and it also restores obfuscated data if requests are sent to trusted domains. Some extensions heavily rely on the communication with their own servers to demonstrate features, for example, an extension that synchronizes users' bookmarks across multiple browser instances requires saving un-obfuscated bookmarks to its server. To add this domain to the trusted list, extension developers need to explicitly declare this specific domain in *manifest.json* file, and users' approvals are also required at the installation time for iObfus to trust this domain. Trusted domains also include the resource URI of extension such as "chrome-extension://<extension-id>".

To capture all cross-origin XHRs, iObfus instruments both the *open()* and *send()* methods of XMLHttpRequest class because sensitive information can either be leaked in the parameters of the target URLs or in the body of *send()* method. To restore sensitive information if necessary, iObfus first determines what domain each specific XHR is sent to. If the request URL matches any one of trusted domains, a de-obfuscation algorithm is then applied to the parameters and the body of XHRs in an effort to reverse the transformation done in the Obfuscator. iObfus also de-obfuscates messages that are written to disk via *LocalStorage*.

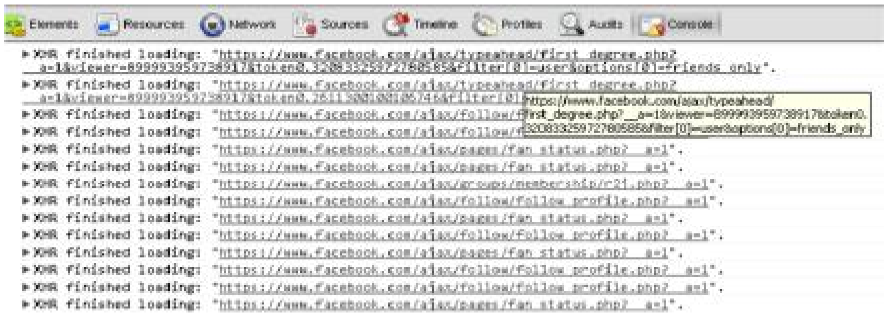
4 Evaluation

To evaluate the effectiveness of iObfus, we first test whether the prototype could defeat some known attacks via malicious extensions and then we assess iObfus's capability to protect users' sensitive information from leaking. At last, the performance overhead introduced by iObfus is studied.

4.1 Mitigate attacks that hijack Social Media accounts

Many rogue extensions that hijack Social Media accounts manifest similar attack behaviors. They are either derived from the same open-source attack toolkit [2] or its variants. Such rogue extensions include Adobe Flash Player 12.1.102.55, Business Adobe Flash Player, Chrome Guncellemesi, Facebook Black, etc. They are briefly described in Appendix C.

We have tested all these social hijacking extensions in our experiments and iObfus can defeat all of them. Figure 2 shows a screenshot when Business Adobe Flash Player was tested in our experiment. Basically, we installed the malicious extension, landed to Facebook.com website and signed in as the test user "iObfus Leakage". The Monitor of iObfus accurately detected the execution of Content Script injected by rogue extension to Facebook.com, and then the Obfuscator processed the DOM document before it was accessed by the Content Script. We observed that the "*c_user=100006040261082"* in Cookie (the Facebook user id of "iObfus Leakage") was replaced with its syntactic equivalence by our obfuscation algorithm. Moreover, since privacy rules defined in section 2.2 contain anti-forgery token keyword *"name=fb_dtsg"*, the value of token "AQBtosAv" was also obfuscated. Hence, the stealthy actions performed by Business Adobe Flash Player failed due to invalid cookie/anti-forgery token as shown in Figure 2.



```
▸XHR finished loading: "https://www.facebook.com/ajax/typeahead/first_degree.php?
  __a=1&viewer=899993959738917&token0.32083325972780585&filter[0]=user&options[0]=friends_only".
▸XHR finished loading: "https://www.facebook.com/ajax/typeahead/first_degree.php?
  __a=1&viewer=899993959738917&token0.26113001001067464&filter[0]
▸XHR finished loading: "https://www.facebook.com/ajax/follow/f
▸XHR finished loading: "https://www.facebook.com/ajax/follow/f
▸XHR finished loading: "https://www.facebook.com/ajax/pages/fan_status.php?__a=1".
▸XHR finished loading: "https://www.facebook.com/ajax/pages/fan_status.php?__a=1".
▸XHR finished loading: "https://www.facebook.com/ajax/groups/membership/r2j.php?__a=1".
▸XHR finished loading: "https://www.facebook.com/ajax/follow/follow_profile.php?__a=1".
▸XHR finished loading: "https://www.facebook.com/ajax/pages/fan_status.php?__a=1".
▸XHR finished loading: "https://www.facebook.com/ajax/follow/follow_profile.php?__a=1".
▸XHR finished loading: "https://www.facebook.com/ajax/pages/fan_status.php?__a=1".
▸XHR finished loading: "https://www.facebook.com/ajax/follow/follow_profile.php?__a=1".
▸XHR finished loading: "https://www.facebook.com/ajax/follow/follow_profile.php?__a=1".
▸XHR finished loading: "https://www.facebook.com/ajax/pages/fan_status.php?__a=1".
```

```
https://www.facebook.com/ajax/typeahead/
first_degree.php?__a=1&viewer=899993959738917&token0.
32083325972780585&filter[0]=user&options[0]=friends_only
```

Fig. 2. Failed stealthy actions of Business Adobe Flash Player

4.2 Protect sensitive information from leaking

We test iObfus prototype against 20 popular extensions from Chrome Web Store. They are shown in Table 4 in Appendix D. The experiment results show that the obfuscation of sensitive information does not hinder the normal execution of most extensions with the exception of several extensions whose features are built on browser user data such as bookmarks or browsing history. These include *bookmark sentry, Xmarks Bookmark Sync* and *PanicButton*. We also observed that 7 of 20 extensions in our study do not initiate any cross-origin XHRs. For extensions that do make cross-origin XHRs, the sensitive information classified in section 2.2 was properly obfuscated before it was read by extensions. Some extensions such as *Google Translate* and *SpellChecker*, did leak obfuscated information via XHRs, but it was not comprehensible to attackers.

However, in our experiments we also noticed that iObfus blocked certain features of extensions that heavily depend on interactions with their own servers, for instance, *Similar Sites Pro* and *WOT*. This is because by design the De-obfuscator only attempts to restore the obfuscated data when the request was sent to safe origins declared specifically by developers, but in reality extension developers often specify excessive web origins with wildcards such as http://*/*. Thus to ensure the data restoration, a set of whitelisted origins are required to be listed in *manifest.json* file. After adding the whitelist, the iObfus works with the browser seamlessly.

4.3 Performance evaluation

We also evaluate the performance of iObfus prototype. The browser version is Chromium 25.0.1347.0 and our test platform is an Intel Core2 Quad 2.66GHz machine with 8GM memory running 64-bit Windows Server 2008R2. The SunSpider 1.0, V8 JavaScript benchmark suites 7.0 and Browsermark 2.0 online tools are used to measure the performance of an iObfus-enabled browser versus unmodified browser. The final result is averaged over 5 runs and shown in Table 2. Compared to the unmodified browser, the performance overhead introduced by iObfus is indeed negligible.

Table 2. Performance comparison between iObfus and unmodified browser

Benchmarks	iObfus	Unmodified browser	Overhead percentages
SunSpider 1.0	393ms	387ms	1.55%
V8 JavaScript benchmark suites 7.0	9422pts	9736pts	3.33%
Browsermark 2.0	4431pts	4695pts	5.96%

5 Related work

Browser extensions can pose significant threats to the security of the browser platform and privacy of browser users [24]. Vulnerabilities in extension platforms have been investigated [6][20], and attacks launched via malicious extensions have been found and reported [25].

Google Chrome has enforced several security policies to protect the browser from attacks via browser extensions. The security model of Chrome is found to be very effective against benign-but-buggy extensions [6], however, it does not consider threats from malicious extensions. Liu et al [7] demonstrated several possible attack scenarios that be achieved by malicious extensions including email spamming, DDOS attacks, password sniff, etc. A refined extension security framework has also been proposed with micro-privilege management and fine-grained access control to DOM elements. Compared to this work, iObfus focuses on defeating the most common and dangerous information leakage attack so that we consider not only the DOM elements in web pages but also browser user data as leakage sources.

Several capacity leaks have been found in JetPack[9], the new Firefox extension framework, via a thorough static analysis [21], many of which can be utilized by attackers to steal user sensitive information. Static analysis techniques are utilized to analyze JavaScript-based extensions. For example, VEX [22] applied a high-precision static information analysis on JavaScript code to identify potential security vulnerabilities in browser extensions automatically. Gatekeeper [23] is another static analysis framework that enforces the security and reliability policy for JavaScript program.

A number of researchers also explored the use of information flow for browser extension security. SABRE [16] is a framework that analyzes browser extensions via tracking in-browser information flows. Djeric et al [8] proposed a framework that is capable of tracking taint propagation at runtime not only in the script interpreter but also in browser's native code. Compared to static techniques, dynamic information flow techniques usually introduce significant performance and/or memory overhead. Our work combines static analysis with dynamic JavaScript instrumentation. iObfus performs a static analysis to mark sensitive DOM elements first, and obfuscates the source or inspects the sink of leakage at runtime. The performance overhead is minimal when compared with traditional dynamic information flow approaches.

6 Conclusion and Future work

Attacks via web browsers pose immense threats to Internet users as web browsers are the most commonly used platform for web surfing. Despite various efforts made, attacks via browser extensions are continuously emerging. In this paper, we seek to protect Internet users from information leakage via browser extension attacks. For this purpose, we have designed and implemented a system called iObfus that can seamlessly work with Chrome. The core of iObfus is to obfuscate sensitive information when there is IO operation pending. We have built a proof-of-concept prototype on Chromium project. Our experiments show that iObfus can effectively mitigate the information leakage threats without degrading users' browsing experience.

To bypass iObfus, attackers could deliberately devise a malicious extension that performs a transformation of sensitive data so that it can circumvent the pattern matching when we inspect network messages. In our next step, we will design and implement new techniques to overcome these attacks.

References

1. Firefox web browser, http://www.mozilla.com/en-US/firefox/firefox.html
2. Qhaoser Hq, an open-source attack toolkit for Facebook. http://userscripts.org/scripts/review/140659
3. Chrome browser features, https://www.google.com/intl/en/chrome/browser/features.html
4. A. Barth, C. Jackson, C. Reis, and T. G. C. Team. The security architecture of the chromium browser. In Stanford Technical Report, 2008.
5. A. Barth. More secure extensions, by default. http://blog.chromium.org/2012/02/ more-secure-extensions-by-default.html, February 2012.
6. N. Carlini, A. P. Felt, and D. Wagner. An Evaluation of the Google Chrome Extension security architecture. In Proc. of the 21st USENIX Security Symposium, 2012.
7. L. Liu, X. Zhang, G. Yan, and S. Chen. Chrome Extensions: Threat Analysis and Countermeasures. In Network and Distributed System Security Symposium (NDSS), 2012.
8. V. Djeric and A. Goel. Securing script-based extensibility in web browsers. In Proc. of the 19th USENIX Security Symposium, 2010.
9. Jetpack, https://jetpack.mozillalabs.com/
10. A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In Proc. of Network and Distributed System Security Symposium (NDSS), 2010.
11. A. P. Felt, K. Greenwood, and D. Wagner. The Effectiveness of Application Permissions. In USENIX Conference on Web Application Development (WebApps), 2011.
12. Chrome extensions flaw allows password theft. http://www.pcpro.co.uk/news/security/359362/chrome-extensions-flaw-allows-password-theft
13. Chromium blog. A year of extensions. http://blog.chromium.org/2010/12/year-of-extensions.html
14. C. Wuest and E. Florio. Firefox and Malware: When Browsers Attack.http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/firefox_and_malware.pdf, 2009
15. F. Assolini. Think twice before installing Chrome extensions. http://www.securelist.com/en/blog/208193414/Think_twice_before_installing_Chrome_extensions
16. M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In Proc. of Annual Computer Security Applications Conference, 2009.
17. Rogue Chrome Extension racks up Facebook "likes" for online bandits. http://www.pcworld.com/article/2028614/rogue-chrome-extension-racks-up-facebook-likes-for-online-bandits.html
18. Health information privacy, http://www.hhs.gov/ocr/privacy/
19. Chesapeake irb, http://chesapeakeirb.com/.
20. 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.
21. R. Karim, M. Dhawan, V. Ganapathy, and C. Shan. An Analysis of the Mozilla Jetpack Extension Framework. In Proc. of the 26th European Conference on Object Oriented Programming(ECOOP),2012
22. S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. In Proc. of the 19th USENIX Security Symposium, 2010.

23. S. Guarnieri and B. Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In Proc. of the 18th conference on USENIX Security Symposium, 2009.
24. D.M. Martin Jr., R.M. Smith, M. Brittain, I. Fetch, H. Wu. The privacy practices of web browser extensions. Communications of the ACM, 2001.
25. Facebook scammers host Trojan horse extensions on the Chrome web store. http://www.pcworld.com/article/252533/facebook_scammers_host_trojan_horse_extensions_on_the_chrome_web_store.html
26. K. Kotowicz and K. Osborn. Advanced Chrome extension exploitation leveraging API powers for better evil. Black Hat, USA, 2012

Appendix A: Security model of Chrome extension

a. Chrome extension architecture

Chrome uses a multi-process architecture, where the browser core process runs in the privileged mode to access system resources and performs I/O tasks and the renderer process is responsible for displaying web content. Figure 3 sketches the extension in the Chrome multi-process architecture. The single instance of the browser core process handles the browser UI and manages all tab and plugin processes, while each renderer process corresponds to a single tab in the browser and runs in a sandboxed environment. To perform any task that needs additional privileges, the renderer process simply sends messages to the browser core process via IPC.

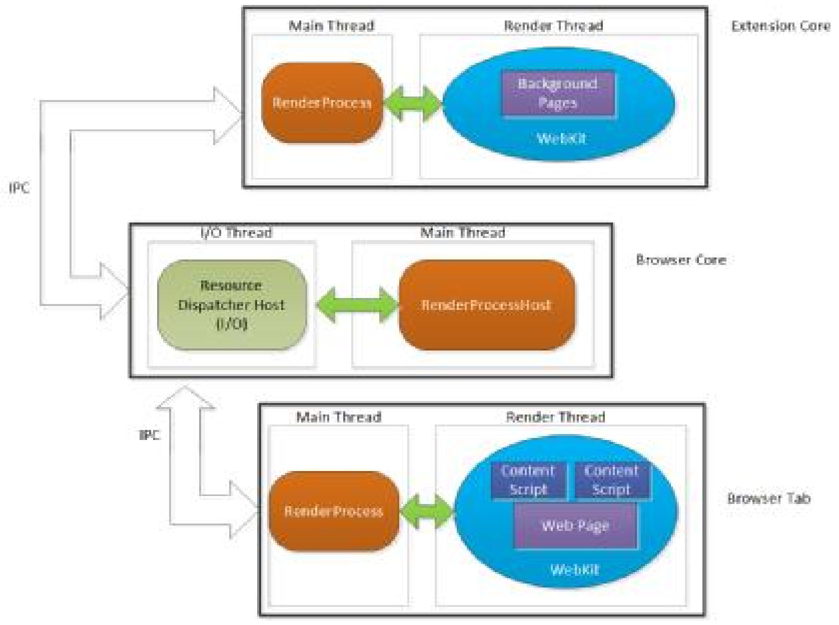


Fig. 3. Chrome multi-process architecture

Chrome also relies on various extensions to extend its functionality. Chrome extension consists of Content Scripts and extension core components. Figure 4 shows an example. A Content Script is a piece of JavaScript code that can be injected into a web page before/after the page is loaded. It is executed in the same process of the tab but in its separate JavaScript engine (called isolated world). Content Scripts cannot access any objects except for the DOM tree of the web page and cannot use any Chrome extension APIs. To communicate with the extension core or Content Scripts across tabs, a Content Script relies on the message passing mechanism of Chrome's inter-process communication (IPC). The extension core contains background web pages and their associated JavaScript code, and it runs in a separate sandboxed renderer process and has no privileges to access system resources or perform I/O. The message passing mechanism is also needed by the extension core to dispatch I/O tasks to the browser core process. Optionally, an extension can have binary code such as NPAPI plugins, which have the same set of privileges as the browser. Note that binary plugins are native executable and not protected by the Chrome extension security framework so we do not include these in our research.

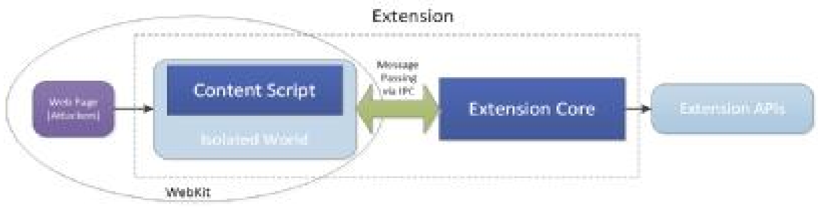


Fig. 4. Chrome Extension Security Architecture

b. Security model of chrome extension

The goal of Chrome extension security architecture is not to defend the browser process against malicious extensions but to protect benign-but-buggy extensions from being compromised. The most common attack against Chrome extensions is through malicious JavaScript codes that are either bundled with web pages or fetched from the network. Thus the security model is an effort to defeat attacks launched from malicious web pages that target vulnerabilities of buggy extensions. To minimize the potential damages caused to the browser process if the extension is exploited, Chrome also uses a multi-component architecture with fine-grained privilege separation strategies.

The security features of Chrome extension model can be summarized as four security mechanisms, and each one further separates privileges that are already refined by the previous mechanism.

1. **Permissions.** The capability of Chrome extension is bounded to its granted permissions. A Chrome browser defines a comprehensive list of permissions that govern access control to Chrome extension APIs and web contents. Each extension is required to declare the permissions it requires in a manifest.json file, as part of the

extension package. The rule of thumb recommended by Google is to request as least permissions as possible. As a security countermeasure to encourage developers to have their extensions scrutinized by the Chrome web store, from Chrome version 21 Google disabled the “Easy install” feature that allows automatic installation of extensions from unknown sources downloaded via a hyperlink.

2. **Privilege separation between extension components.** Different sets of privileges are assigned to the two types of extension components: the Content Script and the extension core. By design, these two components are isolated from each other and are running in separate processes. The Content Script can directly interact with the web page it was injected to, but it cannot use extension APIs to access browser user data. The extension core obtains all privileges requested by extensions but it has to rely on the Content Script to gain accesses to web contents. The communications between the Content Script and the extension core are through exchanging messages over an authenticated IPC channel. The purpose of this privilege separation design is to protect the extension core from attackers in case that the Content Script has been compromised by a malicious web site. Because of the low privilege assigned to the Content Script and the complications to extend attacks over IPC, damages to the extension core are usually controllable.
3. **Isolated runtime environments between the Content Script and associated web page’s JavaScript code.** Although the Content Script is executed in the same renderer process as the web page and shares DOM objects with the web page’s JavaScript, it cannot exchange pointers with JavaScript codes. This is because the Content Script is executed in a separate JavaScript engine (i.e., isolated world), thus it has its own heap and a separate copy of DOM objects. Consequently, it makes object tempering of Content Scripts more difficult.
4. **Content Security Policy (CSP) for extensions.** CSP is a declarative policy that allows web application developers to inform clients what types of scripts can run on its web page. CSP is introduced by Chrome to mitigate cross-site scripting attacks, because the execution of scripts can be blocked if a violation is detected. By default, CSP disables eval and related functions, and it also disables inline scripts. Only scripts and object resources from the extension’s package can be loaded. This will ensure that the extension will only execute user-approved code, not the malicious code redirected by some network attackers. The default policy can be relaxed to a limited extent by whitelisting secure origins if a user has a need for external JavaScript or object resources. CSP can also restrict the use of cross-origin HttpRequest, and iframes.

Appendix B: List of extension APIs that access browser user data

The data in the fourth column of the table 3 are required to be properly processed before they enter the isolated world of the Content Script.

Table 3. List of Extension APIs that access browser user data

Extension API name	Meth-ods/Property	Return Value (property/type)	Taint source
Bookmarks	get getChidlren getRecent getTree getSubtree	Bookmark-TreeNode	url, title
contentSettings	get getResourceI-dentifier	ResourceIdenti-fier	Id
Cookies	get getAll getAllCook-ieStore	Cookie	value, domain, path, storied
History	search getVisit	HistoryItem VistItem	url, title, lastVistTime
pageCapture	saveAsMHTML	MHTML	details(object)
Permissions	getAll contains	Permissions Boolean	permissions, origins
pushMessage	getChannelId	ChannelIdResult	channelId
Storage	get getBytesInUse	Sync Local	Items(object)
Tabs	get getCurrent query	Tab	url, title
Topsites	Get	MostVistedUrl	url, title
Windows	get getCurrent getLastFocused getAll	Window	Tabs

Appendix C: List of malicious extensions that hijack Social Media accounts

- **Adobe Flash Player 12.1.102.55**: The extension disguises itself as the legitimate Adobe Flash Player extension. After installation it gains complete control of victim's Facebook profile and can spread spam messages on Facebook, "like" pages on behalf of victim, etc. The rogue extension spread quickly in Brazil and other Spanish speaking countries before it was detected by Kaspersky Lab researchers [15] and removed from official Chrome Web Store in March 2012.

- **Business Adobe Flash Player**: A variant of abovementioned Adobe Flash Player 12.1.102.55. Attackers keep uploading new variants of Facebook hijacking extensions to Chrome Web Store This extension was removed from Chrome Web Store shortly after it was reported by Bitdefender researchers in February 2013.
- **Chrome Guncellemesi**: Victims are enticed to click on a malicious link embedded in the scam email or messages. The link redirects Facebook users to a web page in Turkish that urges them to download a bogus Chrome update and install the “Chrome Guncellemesi”(Chrome Update) extension. The malicious extension then can collect cookies or interacts with Facebook without users’ consent.
- **Facebook Black**: Facebook users are tricked to open “Facebook Black” landing page, where they are prompted to download the extension. After installation the Facebook Black extension fetches two JavaScript files. One spreads the scam by creating an empty Facebook page on victim’s account with an *iFrame* that redirects users to Facebook Black landing page. The other one is used to present victims a set of survey scams for monetary purposes.

Appendix D: 20 popular extensions in our study

Table 4. Experiment results of 20 popular extensions

Name	Description	Functioning in iObfus	Has Cross-origin XHRs	Leakage mitigated
AdBlock 2.5.63	Blocks ads all over the web	ü	û	–
Google Mail Checker 4.4.0	Displays the number of unread messages in your Google Mail inbox.	ü	ü	–
Stylish 1.1	A user styles manager that lets you easily install themes and skins for Google, Facebook, etc.	ü	ü	–
FastestChrome Browse Faster 7.1.7	Get quick definitions, auto-load next pages, search faster, and more	ü	û	–
Bookmark Sentry 1.7.13	A bookmark scanner that checks for duplicate and bad links.	û	–	–
Google Voice 2.4.1	Make calls, send SMS, preview Inbox, and get notified of new messages.	ü	û	–
Webpage & WebCam Screenshot 8.0	Capture whole page, save PNG, edit, annotate and share to your favorite social network	ü	û	–
Google Trans-	Translates entire webpages into	ü	ü	ü

late 1.2.4	a language of your choice with one click.			
Turn Off the Lights 2.2	The entire page will be fading to dark, so you can watch the video as if you were in the cinema	ü	û	–
SpellChecker 2.76	Prevent spelling, grammar and punctuation mistakes when you write emails and post to social media sites	ü	ü	ü
Xmarks Bookmark Sync 1.0.24	Backup and sync your book-marks, passwords and open tabs across computers and browsers.	û	–	–
SmartVideo for YouTube 0.9926	Provides 'Smart Buffer' for slow connections; auto loop; buffer preferences; quality selection and more	ü	û	–
WOT 1.4.12	Helps you find trustworthy websites based on millions of users' experiences	ü	ü	–
Google Chrome to Phone Exten-sion 2.3.1	Enables you to send links and other information from Chrome to your Android device	ü	ü	–
PanicButton 0.14.2.2	Hide all your tabs at once with one single button and restore them later	û	–	–
Google Dic-tionary 3.0.17	View definitions easily as you browse the web.	ü	ü	ü
Amazon 1 Button App for Chrome 3.2013.530.0	Get special offers and features from Amazon	ü	û	–
Google Quick Scroll 2	Let you jump directly to the relevant bits of a Google search result	ü	ü	ü
Similar Sites Pro 3	Instant access to the best sites related to the one you are browsing	ü	ü	–
Fabulous 27.2	Customize your Facebook with this free app. Block ads, change colors, zoom photos and more	ü	ü	–