

Analysis Notes

Installer (Plug-ins installation)

This context covers issues concerning plug-ins installation, from parsing the plug-in's *configuration file* (in order to extract the plug-in's metadata, resources, permissions, *etc*) to calls to the initialization code provided by the *plug-in*.

Problem(s)

Tag #2: Incorrect user notification of plug-in permissions

Problem:

Alternative title: Improper UI warning of the permissions requested by a plug-in

Plug-ins can request access to certain data/functionality provided by the application host. In such scenario, users need to be informed about which permissions are being requested by the extension. This problem occurs as a result of incorrectly displaying (or not displaying at all) the list of data/functionality that will be accessed by the plug-in.

Mitigation:

Similar to Tag#3 (shown below), we can design a *Central Install Point* that displays all the permissions being requested by the plug-in.

Consequence(s):

- **Data leakage & Privilege Elevation:** a potentially malicious extension can be installed with more permissions than users through of. This can result in the plug-in being able to access sensitive data (data leakage) or perform unauthorized activities (privilege elevation).

Some observed examples:

- CVE-2010-3417 [Chrome]: It does not show to the user that the extension is requesting access to the extension history, which allows attackers to obtain potentially sensitive information via unspecified vectors.
- CVE-2013-0924 [Chrome]: Chrome does not give warnings to users for file permissions obtained by extensions and the API permissions does not look at the value in the checkbox for file-permissions.

Tag #3: Bypassing user notification for plug-in installation

Problem:

This problem occurs when the application breaks the following invariant that must never be breached is: "*all extension installs, no matter how initiated must be mediated by a browser dialog*".

Mitigation:

One way to mitigate the problem is to have a *Central Install Point* in which all the installation requests are guaranteed to go through this component. This component ensures that the install process of an extension is designed to prompt the user asking for permissions to proceed and listing all the relevant permissions of the extension.

Consequence(s):

- **Arbitrary code execution:** a potentially malicious extension can be installed without user awareness and permission.

Some observed examples:

- CVE-2015-4498 [Firefox]: By design, if a direct link to the extension (its XPI file) is copied & pasted into the address bar, Firefox will install the plug-in skipping user confirmation. Although this makes sense (since the end-user was the one who requested the URL), attackers could leverage this design decision to silently install their malicious extensions. In short, they would create a Web page that has a direct link to the XPI file (naming the extension with the same name of any well-known benign extension say "Zotero extension"). Once user clicks on it, the link will force the XPI file to be downloaded and installed and will also redirect to the home Web page of the real extension (e.g., The Zotero's actual plug-in Web page). This way, the extension is silently installed and the user is tricked into believing that he/she installed zotero extension, when, in reality, he/she installed a compromised extension.
- CVE-2011-2358 [Chrome]: By design, they were downloading the crx extension file without prompting the user
- CVE-2011-3055 [Chrome]: Not prompting to the user during install
- CVE-2011-3898 [Chrome]: Bypassing permissions from Java applets
- CVE-2011-3001 [Thunderbird]: pressing the enter button allows arbitrary plugin installation

Tag #5: Code injection to Plug-and-Play Environment via the plug-in's configuration file(s)

Problem:

Alternative title: Improper prevention of code injection in the plug-ins configuration file(s).

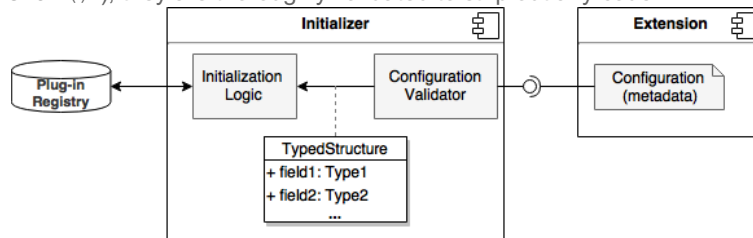
This problem occurs when the host application does not block code that is injected in the plug-in's configuration file.

Researcher's note: this can be merged with Tag #6 (both tags are all about lack of sanitization of the manifest file)

Mitigation:

Configuration files needs to be thoroughly validated after it is parsed. The host application should be careful in not allowing that the configuration file inject executable code to conduct malicious activities.

This can be mitigated through a *Validation Interceptor* pattern in which the raw data contained in the configuration file is converted to a *typed data structure*. Before the fields in this typed data structure are passed to "sink" methods (i.e., methods that perform command execution such as "eval()"), they are thoroughly validated to strip out any code.



Consequence(s):

- **Arbitrary code execution:** a configuration file with inject code can lead to arbitrary code execution.

Some observed examples:

- CVE-2011-2785 [Chromium] - While loading the extension, the extension could inject javascript code in its manifest fields in order to install another extension.

Tag #6: Improper parsing of the plug-in's configuration file

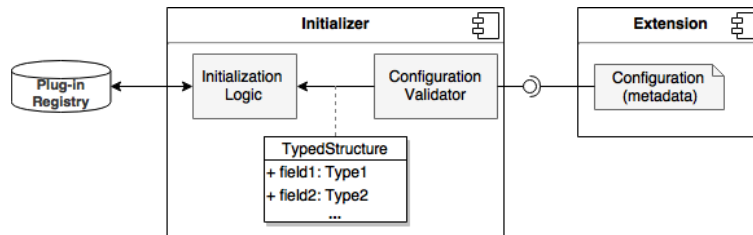
Alternative Title: Assuming the plug-in's configuration file is well-formed.

Problem: Configuration files have a well-defined structure detailed in the application's documentation. Although most of plug-ins will strictly follow such guidelines, a malicious extension or a faulty benign plug-in may create a malformed configuration file, tampering with the parsing mechanism of the host application.

Researcher's note: this can be merged with previous Tag #5

Mitigation: The same Mitigation to [Tag #5](#) can be fine-tuned to also mitigate this problem. This problem can be mitigated through a Validator mechanism. To do so:

- Establish a defined configuration schema that describes the expected data types in each field within the configuration file (simple - Integer, string, etc / complex: objects, URLs, etc)
- Define a *validator* for each data type
- Create a typed data structure that aggregates all these fields (e.g., a class)
- Create an interceptor in the installer component that validates each field (using the implemented validators). Then, if the validation was successful, it converts each raw field in the configuration file to the corresponding field in the typed data structure.



Consequence(s):

- **Availability:** assuming the manifest file is well-formed can lead to a crash (memory corruption) of the parsing mechanism.

Some observed examples:

- CVE-2008-2807 [Firefox]: Faulty .properties file results in uninitialized memory being used - assuming utf8 encoding when it shouldn't be assumed Internationalization

- CVE-2014-3170 [Chrome]: NULL bytes allowed in hosts permission allowing for obtaining more permissions for extensions.
- CVE-2011-0779 [Chrome] Accepting mal-formed extensions (assuming they will follow the guidelines). If an extension provides an empty signature, it was resulting in a crash on the browser.
- CVE-2013-0831 [Chrome] Directory Path Traversal in loading extension's resources

Tag #7: Improperly checking the origin of an install request

Problem: This problem is a result of not checking (or improperly) checking the source of an install request. This improper check of who initiated the install request can result in unintended plug-ins to be installed.

Mitigation: Whenever a new install is triggered, the application host needs to check from where this request came from. Ideally, a plugin-based architecture should have only well-defined install points that can trigger the installation, avoiding that malicious code or other vectors silently install a plug-in.

Consequence(s):

- **Data leakage & Privilege Elevation:** a potentially malicious extension originated from an unintended source can be installed in the user's machine.

Some observed examples:

- CVE-2015-0812 & CVE-2016-1948 [Firefox]: HTTPS scheme not checked which causes the possibility for MITM when installing themes.
- CVE-2011-2370 [Firefox]: The whitelist where xpinstall can be triggered is not enforced
- CVE-2016-1640 [Chrome]: The Web Store inline-installer implementation does not block installations upon deletion of an installation frame, which makes it easier for remote attackers to trick a user into believing that an installation request originated from the user's next navigation target via a crafted web site.

Tag #27: Execution of user-defined blocked plug-ins

Problem:

Context: The application may allow the user to block certain plug-ins from being executed.

Causes: This problem is caused by not correctly checking the user-defined list of blocked plug-ins. This results in a bypass of this blocking mechanism, allowing the plug-in to execute normally, and potentially perform malicious activities.

Mitigation:

Consequence(s):

- **Security Bypass:** a potentially malicious plug-in can be executed without the user's consent.

Some observed examples:

-

Sandbox & Plug-in Isolation

It runs the extension's logic code on an isolated and sandboxed environment. It also enforces the permissions specific to the extension (as provided by the initializer).

Problem(s)

Tag #1: Improper Isolation of Objects used in Plugins or Plug-and-play environment

Problem: Improper Isolation Enforcement of Data Structures / Objects used by the core and the plug-ins.

It occurs when the host application does not isolate objects from different compartments

When the host application and plug-ins share the same objects and/or data structures, they may be able to change the properties of these objects/data structures, which can negatively interfere other plug-ins or the host application. These shared objects create a channel that can be used to inadvertently breach information or be used to tamper with other plug-ins or the application host.

Mitigation:

Isolated Object Domains

In short, each plug-in should have its own copies of objects that are used for message passing / implement the plug-in API interface. Then, the host application, manages these objects, ensuring that pointers/references passed to the plug-in are not pointing to objects from the host application or other plug-ins.

This is, indeed, how Chrome designed its architecture. Each extension can use the Javascript language to access the DOM (Document Object Model) of the Web pages. However, to enforce that extensions do not interfere with each other or with the Google Chrome browser, it designed different JavaScript objects per extension (see Figures below that were extracted from Barth *et al* 2009 - *Protecting Browsers from Extension Vulnerabilities*).

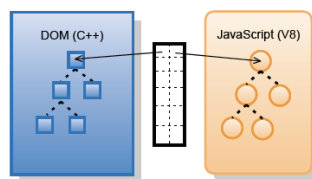


Figure 7. The normal one-to-one relation between DOM implementation objects and JavaScript representations.

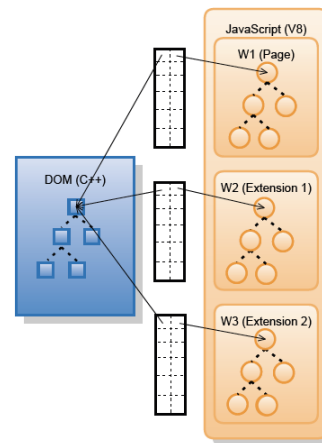


Figure 8. The one-to-many relation caused by running content scripts in isolated worlds.

This design is lacking in Thunderbird and Firefox (Extensions in these systems share Javascript objects). Also, Pidgin, WordPress, OpenMRS, and OfBiz do not have any mechanism to isolate objects from their internal APIs for plug-ins usage.

Consequence:

- **Alter execution logic** - if the objects are shared, extensions may be able to change the properties of this object, which can negatively interfere the behavior of other plug-ins or the host application.
- **Data Leakage** - external plug-ins could use this objects as a mean of obtaining data without any authorization
- **Bypass protection mechanism** - shared objects could be used as “covert” channel for successful attacks
- **Execute arbitrary code** -

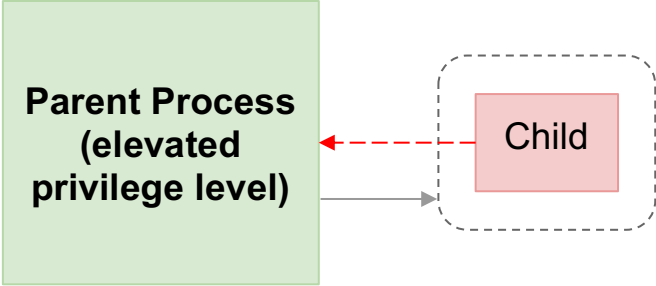
Some observed examples:

- CVE-2006-6499 [Thunderbird] - Although the issue seemed to be just a miscalculation bug, from what I saw in the discussions, if they had used the Chrome's approach of isolating javascript objects, this bug would never be exploited. Why? In Chrome, Javascript objects are different (each plugin has its own JS objects), this means that plugins cannot interfere with each other nor the hosting application.
- CVE-2008-0418 [Thunderbird] - Extensions can traverse the directory and read from sessionstore.js file. Similar to CVE-2006-6499 this vulnerability is possible because they share javascript objects by design.
- CVE-2012-2877 [Chrome] - Extensions leading to a crash in the chromium's kernel process.

Tag #8 - Allowing a sandboxed process to manipulate a higher-privileged process

Problem: The sandboxed child process can manipulate the process handle of the parent process to perform operations at a higher privilege levels, thereby escaping the sandbox.

In a sandboxed environment, the child process (in this case, hosting the extension/plug-in) typically hold a reference (or pointer) to the process handle of the parent. Through this pointer, the extension may leverage OS system calls to breach the sandbox and obtain full access to the parent process.

	
Mitigation: This problem could be avoided by blocking OS system calls. This means, leveraging a mechanism that intermediates any system call between the sandboxed child process and the underlying OS.	
Consequence: <ul style="list-style-type: none"> • Arbitrary code execution • Privilege escalation 	
Some observed examples: <ul style="list-style-type: none"> • CVE-2014-8643 [Firefox]: Sandbox escape through process handle • CVE-2011-3004 [Firefox]: Handling unwrapped windows to higher privileged code • CVE-2011-0076[Firefox]: Plug-in related - elevation of privilege 	

Tag #9 - Improper principal check

Alternative title: Lack of security checks when performing a privileged action Problem: Whenever a new incoming request is made to the sandbox mechanism, the sandbox checks the origin of that request. This problem occurs when the system fails to correctly check the origin of the request.	
Mitigation:	
Consequence: <ul style="list-style-type: none"> • 	
Some observed examples: <ul style="list-style-type: none"> • CVE-2016-2817 [Firefox]: Extensions are allowed to directs tabs to sites (that can be harmful) without permission, elevating their privileges • CVE-2009-2665 [Firefox]: Plugins can cause the wrong security wrapper to be used 	

Tag #13: IPC mechanism passes raw pointers to plug-ins **[MERGED WITH TAG#1]**

Problem: Passing raw pointers to plug-ins allow them to perform unbounded operations, bypassing any security checks. <i>Researcher's note: it is same as Tag #1.</i>	
Mitigation:	
Consequence: <ul style="list-style-type: none"> • Data leakage 	
Some observed examples: <ul style="list-style-type: none"> • CVE 2010 1229 [Chrome] A compromised renderer can pass an arbitrary pointer to the plugin process; this pointer is then dereferenced and manipulated (written) in the plugin process. This could be used to corrupt the plugin process and execute arbitrary code outside the sandbox. • 	

Tag #14: Message passing mechanism does not indicate who sent the message [Same as Tag #9]

Problem: The IPC mechanism sends data to another process without forwarding associated metadata about which process sent that message.

Researcher's side note: this is same as Tag #9

Mitigation:

Consequence:

- Data leakage

Some observed examples:

- CVE-2014-1728 [Chrome]: The exploit is possible because the attacker's renderer process can send a message from a swapped out RenderFrameHost which makes it up into TabHelper. TabHelper isn't keeping track of who sent the message and assumes it's from WebContents::GetRenderViewHost(), which refers to the current RenderViewHost (legitimately for the extension process) and not the swapped out one (in the attacker's process).

Tag #15 Breaches in Plug-ins Encapsulation

Problem: Not all features and code from the application host are meant to be available to plug-ins. This problem occurs when the plug-ins can access these hidden data/functions.

This Tag also encompasses the type of problems in which plug-ins are able to access data or call functions from other plug-ins without being authorized to do so.

Mitigation:

Consequence:

- Execute unauthorized code
- Bypass protection mechanism

Some observed examples:

- [CVE-2011-1123](#) [Chrome]: An extension had some non-public native code (DLLs). Although they have specified on its manifest files that these binaries are meant to be private, other extensions (or Web pages) could directly invoke functions from these DLLs.
- [CVE-2016-1638](#) [Chrome]: Some Web APIs are meant to be kept private and not be accessible (i.e., used by) extensions. However, due to an incorrect implementation of API restrictions, these Web APIs were being called by extensions

Tag #16 Improper object-life management

Problem: Objects from plugins 'talk to' objects from the application core. The issues pertaining to this violation arise because of core applications calling objects in the plugin side that have been destroyed prematurely.

Mitigation: This violation could be avoided by introducing a mechanism that checks certain objects of interest lifetime and notifies the core application on their status.

Consequence:

- Race conditions
- Use-after-free
- Core application crash

Some observed examples:

- CVE-2015-2706 [Firefox]: The application core fails to destroy objects related to a plugin that fails to initialize properly.
- CVE-2011-3107 [Chrome]: Chrome crashing trying to call hasMethod which has a non-valid pointer.

Tag#17 Improper Object Access Control and Compartmentalization Enforcement

Problem: When privileged code uses unprivileged code, the unprivileged code is wrapped to protect the privileged code. However, the wrappers used are improper and elevate the privileges of the unprivileged code.

*Side note: this is a trust boundary problem.
Alternative title: Improper object wrappers*

Mitigation:

Consequence:

- Elevation of privileges
- Arbitrary code execution

Some observed examples:

- CVE-2008-2803 [*Thunderbird, Firefox*]The mozIJSSubScriptLoader.LoadScript function in Mozilla Firefox before 2.0.0.15, Thunderbird 2.0.0.14 and earlier, and SeaMonkey before 1.1.10 does not apply XPCNativeWrappers to scripts loaded from (1) file: URIs, (2) data: URIs, or (3) certain non-canonical chrome: URIs, which allows remote attackers to execute arbitrary code via vectors involving third-party add-ons.

Tag#20 Improper object locking in concurrent threads [does not qualify for being a core category, see note below]

Problem: The application incorrectly locks an object leading to race condition problems and crashes.

Researcher's side note: only one instance (CVE-2016-1650) was found in the analysis, it does not fit our criteria for being a core category

Mitigation:

Consequence:

- Race Condition
- Use-after-free

Some observed examples:

- CVE-2016-1650 - The PageCaptureSaveAsMHTMLFunction::ReturnFailure function in browser/extensions/api/page_capture/page_capture_api.cc in Google Chrome before 49.0.2623.108 allows attackers to cause a denial of service or possibly have unspecified other impact by triggering an error in creating an MHTML document that stems from not locking the object properly.

Tag #23 - Lack of a sandboxing mechanism

Problem: A lack of a sandbox (or also known as jail) allow the plug-in to perform unbounded operations, accessing all system resources (files, databases, etc) in an unpredictable fashion.

Mitigation: This problem could have been avoided by creating a jail in which operations are limited to a subset of privileges. Moreover, the code is confined within a specific folder, meaning that it cannot access any other files from the system.

Consequence:

- Arbitrary code execution
- Privilege escalation

Some observed examples:

- CVE-2006-5705 [Wordpress]: WP-DB-Backup plug-in has a directory traversal vulnerability, resulting in read or overwrite of files. A Sandboxing mechanism could have isolated this plug-in and prevented the escape.

Tag #25 - Lack of fine-grained permissions

Problem: Many vulnerabilities observed in Pidgin and WordPress happened because of benign-but-buggy extensions that had

overpermissions. The real problem here is: none of these case studies has a permission-based security model for ensuring that plug-ins only access certain data/functions upon requests.
Mitigation: This problem could have been avoided by following how the other three case studies (Chrome, Thunderbird and Firefox) provides data and functions: - Plug-ins have to explicitly specify the type of data and functions they intend to use on their manifest file
Consequence: <ul style="list-style-type: none"> Privilege escalation
Some observed examples:

Deallocator

Tag #18: Unsafe directory for plugins

Problem: The core application's directory for plugins is predictable and can be overridden by malicious parties.
Mitigation: Create unique directories to save plugin related artifacts.
Consequence(s): Overridden user files
Some observed examples: <ul style="list-style-type: none"> CVE-2005-0578 [Firefox] Firefox before 1.0.1 and Mozilla Suite before 1.7.6 use a predictable filename for the plugin temporary directory, which allows local users to delete arbitrary files of other users via a symlink attack on the plugtmp directory.

Tag #19: Unsanitized plugin data

Problem: The core application interacts with data from the plugins. The problem arises when this data is not properly sanitized. The application host trusts data from the plug-in when it shouldn't because this data is crossing boundaries.
Mitigation: Introduce mechanisms that sanitize the data flowing from plugins to the core application.
Consequence(s): Arbitrary code execution, Denial of service
Some observed examples: <ul style="list-style-type: none"> - CVE-2005-0752 [Firefox]: The Plugin Finder Service (PFS) in Firefox before 1.0.3 allows remote attackers to execute arbitrary code via a javascript: URL in the PLUGINSOURCE attribute of an EMBED tag. - CVE-2013-0896 [Chrome]: BrowserPluginGuest trusts the shared memory region sizes passed in messages from renderers. When the browser attaches to these regions it does not sanity check the region sizes and can be made to write beyond the end of the mapped region. - CVE-2012-5328 [WordPress]: Multiple SQL injection vulnerabilities in the Mingle Forum plugin 1.0.32.1 and other versions before 1.0.33 for WordPress might allow remote authenticated users to execute arbitrary SQL commands.

Event Management

Dispatching events to plug-ins is crucial for an extensible architecture. This is the mechanism to which plug-ins are attached.

Security Problems:

Tag #4: Reentrant Event Callbacks

Problem: Reentrant event callbacks This problem occurs when extensions can interrupt the execution of the event dispatching mechanism before it has finished, resulting in an unpredictable state.
Mitigation: Given that multiple events may arrive and need to be dispatched to many plug-ins, it is important to ensure that the callback mechanism at the application host perform these operations in an atomic fashion. To mitigate the problem: while performing the dispatch of the event to the corresponding extensions this operation is designed to be atomic such that it guarantees the correctness of the initialization (avoid to leave the extension in an invalid state). It can be done so through a locking mechanism.
Consequence(s): <ul style="list-style-type: none">• Availability: it can cause a crash due to this corrupted state
Some observed examples: <ul style="list-style-type: none">• CVE-2016-1635 [Chrome]: Apps & extensions can make the callback routine to be invoked reentrantly, resulting in a crash (use-after-free)• CVE-2013-2912 [Chrome]: The resource tracker tries to use the object which is half destructed.• CVE-2015-6772 [Chrome]: Re-entrancy while attaching a new document in a frame when an old document is being detached (use-after-free)

Tag #12: Events Dispatching without Authorization of Event Subscribers/Listeners

Problem: Dispatching events without considering permissions.
Mitigation: This problem exists when the software application dispatches events to all the plug-ins that registered to that event without checking whether the plug-in is authorized to listen to that event.
Consequence: <ul style="list-style-type: none">• Data leakage
Some observed examples: <ul style="list-style-type: none">•

Plug-in Update

Tag #22: Elevation of privilege through plug-in update

Problem: A plug-in is able to elevate its privileges after an update. <i>Alternative title:</i> Not preventing privilege elevation upon update
Mitigation: Each plug-in provides a list of privileges in which the host application verifies before install. However, these permissions need to be enforced at lifetime: once a user confirms a set of permissions, these cannot be changed after an update, in order to make an extension with higher privileges. This problem occurs when the application host breaks the invariant that permissions should not be elevated after an update.
Consequence(s): <ul style="list-style-type: none">• Privilege escalation
Some observed examples: <ul style="list-style-type: none">• CVE-2013-2868 [Chrome]: Chrome allows for extensions to obtain permissions after update to install plugins and execute code. The mistake is in the browser part.

Maintaining Plug-in Registry

Tag #18: Unsafe directory for plugins

Problem: The core application's directory for plugins is predictable and can be overridden by malicious parties.
Mitigation: Create unique directories to save plugin related artifacts.
Consequence(s): <ul style="list-style-type: none">• Overwrite user files
Some observed examples: <ul style="list-style-type: none">• CVE-2005-0578 [Firefox] Firefox before 1.0.1 and Mozilla Suite before 1.7.6 use a predictable filename for the plugin temporary directory, which allows local users to delete arbitrary files of other users via a symlink attack on the plugtmp directory.

Tag #11: Extraction of the add-on bundle with world-readable/writable permissions

Problem: Extraction of the add-on executable with world-writable permissions to an unsafe folder. In general, plug-ins are released as software bundles (zip files, archive, etc) that are extracted by the application. This problem occurs when the application extracts these bundles using world-readable (or writable) permissions (e.g. 777 permissions). This allows other individuals to tamper with the contents of the plug-in.
Mitigation: One way to mitigate the problem is to extract the software bundle to the application's dedicated folder, with a lower permission.
Consequence(s): <ul style="list-style-type: none">• Modify Plug-in Data• Alter plug-in functionality• Execute unauthorized code
Some observed examples: <ul style="list-style-type: none">• CVE-2013-0798 Firefox - Add-ons in the Firefox Android have permissions set to readable and writable (777), thereby malicious applications can change them• CVE-2004-0906 Thunderbird - The extraction of extensions files was made with wrong file permissions, creating world-readable/writeable files. This allows a trojan to modify a benign application for malicious purposes.• CVE-2008-6811 Wordpress - Plugin allows to upload executable files and modify certain directories to call them•

Blacklists

One design consideration for using blacklists is to prevent malicious extensions to be installed and loaded in the user's Web browser. For this blacklist to be effective it has to be updated continuously.

Problem(s):

Description: When the blacklist is updated through a transmission over the network (e.g. HTTP request) it can be subject to "man-in-the-middle" attacks that would be able to create a fake blacklist. One potential violation is to not prevent MITM attacks that tamper with the blacklist file.

Mitigation: The blacklist manifest is fetched over https and includes a sha256 hash - when we fetch the blacklist content over http we verify that the content's hash matches that. In short, it is a combination of "Verify Message Integrity" and "Encrypt Data" tactics.

Affected Security Property(ies):

- Integrity - the faked blacklist could allow malicious extensions to be installed in the user's machine

Observed Example:

- In Chromium, implemented in the `ExtensionUpdater::ProcessBlacklist` (briefly discussed in bug [#108648](#))

Description: The source of the blacklist file has to be always available to be fetched. One potential violation is that an extension could prevent the application from receiving the blacklist update. This prevention is done through injecting malicious code in their logic that blocks any attempt to request to the specific blacklist URL.

Mitigation: The object that is used to retrieve the blacklist from the Web server is not the same object that it is used by extensions to perform Web requests. This creates isolated spheres.

Affected Security Property (ies):

- Availability - the code blocks requests to the blacklist URL.

Observed Example(s):

- CVE-2011-3049: An extension could add code in its background script that registers a listener that is triggered before any Web request (`chrome.webRequest.onBeforeRequest.addListener()`). Basically, the listener will check the URL of the request and if it verifies that it is the blacklist URL, then it blocks the request, which prevents the update.