## **Search Query**

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Search Query** | **Total** | **Source** |
| Q-11 | ((plugin OR plug-in OR extension) AND (vulnerability OR vulnerabilities)) | 243 | IEEE Explore |
| Q-12 | ((plugin OR plug-in OR extension) AND (vulnerability OR vulnerabilities)) | 3,503 | ACM Digital Library |
| Q-13 | ((plugin OR plug-in OR extension) AND (vulnerability OR vulnerabilities)) | 6,950 | SpringerLink |
| Q-14 | ((plugin OR plug-in OR extension) AND (vulnerability OR vulnerabilities)) | 357 | Science Direct |
|  | **Total** | **11053** |  |

## **Inclusion/Exclusion Criteria**

|  |  |
| --- | --- |
| **Inclusion Criteria** | |
| I1 | Full Paper |
| I2 | Focused on discussing security problems on a plug-and-play software architecture |
| **Exclusion Criteria** | |
| E1 | Books, position papers, short papers, tool demo papers, keynotes, reviews, tutorial summaries, and panel discussions |
| E2 | Not fully written in English |
| E3 | Duplicated study |
| E4 | Not focused on security problems on a plug-and-play software architecture (Violation of I2) |

## **Included Paper Notes**

## Platform-Based Plug and Play of Automotive Safety Features: Challenges and Directions (Gangadharan *et al*, 2016)

**Summary**

The paper presents a proposal for an infrastructure that would facilitate the deployment of automotive safety features on-demand from a cloud server to the customer’s vehicle, after running feasibility checks. This infrastructure, according to the paper, would be implemented in a PnP environment for automotive safety features. The security angle is limited to ensuring that the security requirements are satisfied and that the features are ensured to be compatible with the rest of the software. In another passage, the paper mentions that a possible attack vector could come from using a malicious PnP application. In order to ensure the security of the system, the paper proposes a new extension to CAN(Controller Area Network) bus which is the most popular internal networking system in this domain. Currently, the bus is shared between safety-critical and non-safety-critical systems. They propose using MAC(Message Authentication Code) on the CAN data with a resource configuration method to ensure the communication is carried out securely *by creating separate message channels based on security level*, i.e. authentication especially for the highly secured channel. This requires foregoing some of the data bits for these authentication processes.

**Analysis Notes / Relevant Excerpts**

“*Another possibility to attack the safety critical components is to use a PnP application that intends to compromise and breach the vulnerability of vehicle systems with malicious purposes*.”

“*Currently, the most popularly used internal networking system is CAN (Controller Area Network bus). Safety-critical software units, such as Engine Control Unit, Transmission Control Units, etc., share the CAN bus with non-safety-critical software components, such as infotainment systems. If a malicious program gets access to the infotainment systems, it will be able to access the data from safety-critical software components over the shared CAN. One of the popular ways to bring security to CAN is the entity authentication between communication nodes using MAC (Message Authentication Code) [23], [24]. Since CAN is originally not designed to account for the security problem [25], some of the data bits for data transmission need to be sacrificed for the authentication. The more secured channel needs more data bits, i.e., the data bits of CAN used for data transmission should be sacrificed to guarantee the highly secured channel. For this reason, the acquisition of the resource for security protocols is critical to guarantee the security of PnP apps. The literature [23], [24] proposes a resource configuration method that guarantees a secure communication using MAC upon CAN data payload. However, it needs to take into account a dynamic security resource allocation for PnP app environment*.”

The paper does deal with plug-and-play architectures, albeit the attention to security is more limited:

"*Another possibility to attack the safety-critical components is to use a PnP application that intends to compromise and breach the vulnerability of vehicle systems with malicious purposes.*"

“*Safety-critical software units, such as Engine Control Unit, Transmission Control Units, etc., share the CAN bus with non-safety-critical software components, such as infotainment systems. If a malicious program gets access to the infotainment systems, it will be able to access the data from safety-critical software components over the shared CAN. One of the popular ways to bring security to CAN is the entity authentication between communication nodes using MAC (Message Authentication Code) [23], [24].*”

To solve this challenge, the paper indicates that “*the acquisition of the resource for security protocols is critical to guarantee the security of PnP apps.*”. Here, the paper discusses that one way to solve the problem is via a secure communication of plug-in resources.

Component-Oriented Monitoring of Binaries for Security (Rajkumar et al, 2011)

**Summary**

This paper proposes a framework (COMB) that implements confining behavior policies at a fine-grained level of components, plugins included. The framework relies on monitoring the binary executable of the program to implement its policies.

**Analysis Notes / Relevant Excerpts**

The paper deals with security policies that confine the behavior of certain components, plugins among them. Even though extensible architecture is not their main focus, they test their policy implemented in a tool in Firefox because of its extensible nature and the third-party add-ons.

From the paper reading we noticed that it discusses aspects from our “*Improper Object Access Control and Compartmentalization Enforcement*”. It also discusses that the host monitoring components for specific things they should not be doing.

This paper overlapped with the following concepts from our theory:

VULNERABILITY TYPE

* Improper object access control in compartmentalized PnP environment

MITIGATION

* Compartmentalization of plug-ins
* Security Policy Enforcement through Object Wrappers: Fine-grained security policy that would limit the access control of plug-ins specifically
* Limit plug-ins exposure to OS processes

The paper also provides mitigations that we haven’t observed from our data which is to limit plug-ins exposure to OS processes as well as runtime monitoring of plug-ins’ behavior (through system-call monitoring policies enforced at runtime).

## MAuth: A Fine-grained and User-Centric Permission Delegation Framework for Multi-Mashup Web Services (Alam et al, 2010)

**Summary**

The paper presents a framework for fine grained permissions between mash-up services in a web application.

**Analysis Notes / Relevant Excerpts**

The paper overlaps with one concept that we observed in our theory (Incorrect user notification of plug-in permissions).

- users are confused about which applications have which permissions or where their data is coming from. This paper seeks to address the problem by providing a permission delegation framework.

A Module System for Isolating Untrusted Software Extensions (Fong and Orr, 2006)

**Summary**

The paper proposes a method to isolate better software extensions in extensible(PnP included) architectures. The method relies on namespaces and name visibility as the access control method. The core application will make use of this method by controlling the granting of access to different names to different extensions, at different levels.

**Analysis Notes / Relevant Excerpts**

**“**ISOMOD employs name visibility control as the sole mechanism for access control. We describe the Java class loading mechanism from the perspective of name visibility control. In programming language terminology, a Java class loader is the mirror [5] of a run-time namespace. Hierarchical organization of namespaces is enabled by the delegation model of class loading [18], in which the names visible in a parent namespace are imported implicitly into a child namespace (Fig. 1). Consequently, the set of names visible in a namespace is the union of (1) the set of names visible in its delegation parent and (2) the set of names that are defined locally. A class may refer to external entities such as other classes and their fields and methods. These external references are resolved in the same namespace in which the referring class is defined. In short, static scoping is enforced.”

“A core application may create namespaces dynamically and impose arbitrary name visibility policies to control whether a name is visible, to whom it is visible, and in what way it can be accessed.”

“In a dynamically extensible software system, the trusted application core is defined in a parent namespace, while child namespaces are created for defining untrusted software

extensions . Core application services are exposed to the extension code through implicit name import from the core application namespace to the extension namespace.”

“An access is composed of three elements: (1) a subject, (2) an object, and (3) an access right.”

Overlapping concepts:

Tag #15 Improper object access control in compartmentalized PnP environment

Tag #8 - Allowing a plug-in to elevate its permission by manipulating (or delegating a task to) a process in the PnP environment that has higher privileges

Mitigations: Isolated Object Domains, Declarative-based request for accessing data/functionality

It also discusses a new mitigation technique that relies on name visibility policy as a method of access control (Named-based access control\*\*)

<< "*present the design of a practical security architecture for dynamically extensible Java applications that is built around a module system called ISOMOD (Sect. 2). In programming language literature, a module system is a facility responsible for managing the visibility of names across namespaces In a dynamically extensible software system, the trusted application core is defined in a parent namespace, while child namespaces are created for defining untrusted software extensions . Core application services are exposed to the extension code through implicit name import from the core application namespace to the extension namespace. ISOMOD is a run-time module system designed for isolating untrusted software extensions.*" >>

## Equipping WAP with WEAPONS to Detect Vulnerabilities: Practical Experience Report (Medeiros et al, 2016)

**Summary**

The paper presents a tool to detect vulnerabilities in applications written in PHP. Among others, the tool was tested on WP Plugins. The analysis was carried out in 115 plugins in WP. The paper found 55 SQLI vulnerabilities and 114 other vulnerabilities in 23 of those plugins. SQLI and XSS vulnerabilities were among the most common ones, with 55 SQLI vulnerabilities and 77 XSS vulnerabilities.

**Analysis Notes / Relevant Excerpts**

"We selected 115 plugins from different tags (arts, food, health, shopping, travel, authentication, popular plugins and others) and distributed by several ranges of downloads, from less than 2000 to more than 500K. The popular plugins fit in this last range, having some of them more than 1M downloads."

" WAPe discovered 153 zero-day vulnerabilities and detected 16 known vulnerabilities. Table VII shows the 23 plugins with vulnerabilities, distributed by 8 classes. The wpsqli weapon detected 55 SQLI vulnerabilities, while the other detectors found the remaining 114 vulnerabilities of the XSS, RFI, LFI, DT, HI and CS classes (last 2 are new)."

"Fig. 5 presents the vulnerabilities detected by class for the 17 web applications and 23 WP plugins. Clearly SQLI and XSS continue to be the most prevalent classes."

Although the paper focuses around a tool to detect vulnerabilities, but has a section that talks about vulnerabilities in WP plugins that are useful. In particular, they talk about an overlapping concept: Unsanitized plug-in data exploiting the host application (this case, WordPress).

## phpSAFE: A Security Analysis Tool for OOP Web Application Plugins (Nunes et al, 2015)

**Summary**

The paper's focus is evaluating the presence of the vulnerabilities(XSS and SQLi) in plugins. The authors of the paper have been asked to develop this tool by Automatic, the developer of Wordpress. In the evaluation they have used WordPress and 35 plugins of theirs to test the tool and found 151 vulnerabilities in 10 plugins in their 2012 version, and 179 vulnerabilities in 7 plugins in their 2014 version.

They also make a point that plugin developers display an inertia in solving plugin vulnerabilities, even after they're known for a while.

**Analysis Notes / Relevant Excerpts**

" phpSAFE is a static code analyzer for detecting XSS and SQLi vulnerabilities in PHP plugins"

"phpSAFE found 151 vulnerabilities related to the use of WordPress objects in 10 plugins of the 2012 version, and 179 vulnerabilities in 7 plugins of the 2014 version. "

"Combining the results of all tools [phpSafe and two other tools used for comparison] we detected 394 distinct vulnerabilities in 2012 versions and 586 in 2014 versions."

Inertia in fixing plugin problems

"*One of the quality assurance activities that should be done while maintaining software during its lifecycle is fixing bugs, giving priority to those that are more critical, like security issues.*

*The vulnerabilities found in the 2012 version of the plugins were initially disclosed to the developers in November 2013 [3]. In the present study we analyzed which of the vulnerabilities found in the 2014 version were among the ones previously disclosed in the 2012 version. We found that 249 (42%) of the vulnerabilities discovered in the 2014 version are among the ones discovered and disclosed to the developers more that one year ago. From those, 59 (24%) are very easy to exploit (through GET,POST or COOKIE manipulation). This is a disturbing result that should raise the awareness of plugin developers, of maintainers of the CMS frameworks, of the site administrators, and of the end users.* "

Tangential connection: talks about plug-in vulnerabilities

After reading and analyzing the paper many times: The paper's focus is evaluating the safety of plugins. However, in the evaluation they've used WordPress and 35 plugins of theirs. They also make a point that plugin developers display an inertia in solving plugin vulnerabilities, even after they're known for a while. Might come handy to argue that architectural solutions are more urgent for pluggable architectures. In the end, this paper discusses the problem we also found “Tag#19 Unsanitized plugin data”.

Cujo: Efficient Detection and Prevention of Drive-by-download Attacks (Rieck et al, 2010)

**Summary**

The paper presents a tool, Cujo, for automatic detection and prevention of drive-by-download attacks that make use of exploits in browsers and their extensions. Thus, this paper overlaps with the problem we also uncovered: Tag #2: Incorrect user notification of plug-in permissions. The consequence of this vulnerability: Drive by download (Silent install of plug-ins).

Ianus: Secure and Holistic Coexistence with Kernel Extensions - a Immune System-inspired Approach (Oliveira et al, 2014)

**Summary**

This paper presents Ianus, a paradigm to protect the OS kernel from malicious extensions. The method proposed makes use of security policies that restrict the number of functions the kernel extensions have access to, isolating them, and also isolating their write access to only their memory and not the kernel. The implementation of the method relies on a higher-priviledged process knowing the boundaries of each module, i.e. extension, their origin, write operations in the CPU and functions which they call. This is related to the tags for origin check and compartmentalization. The implemented method was tested against rootkits that attempt to tamper with the kernel and was successful in stopping them.

**Analysis Notes / Relevant Excerpts**

“In this work a similar paradigm for OS defense is proposed, where the OS plays an active role in controlling the execution of its extensions. The following security policies are adopted: (i) kernel extensions should only interact with the kernel through its exported functions, and (ii) kernel extensions should never directly write into kernel code and data segments, except into their own memory (module address space and its dynamically allocated areas).”

“Controlling the execution of kernel modules lies at the heart of the proposed approach, which relies on: (i) knowledge of boundaries of kernel modules in main memory, (ii) monitoring of write operations in kernel space, (iii) knowledge of the origin of the current instruction being executed at the CPU, and (iv) knowledge of the addresses of functions invoked by modules.”

The paper discusses the Linux kernel extensions. Ianus’ security was analyzed against real rootkits that exercised the following security concerns: (i) tampering with kernel code and data, (ii) tampering with downcall parameters, and (iii) issuing bogus downcalls. False positives were also evaluated with benign kernel modules and drivers.

This paper overlaps with an Improper Isolation of Objects used in Plugins or Plug-and-play environment.

It also talks about a mitigation that is the isolation of object domains.

An Analysis of the Mozilla Jetpack Extension Framework (Karim et al, 2012)

**Summary**

The paper discusses the Firefox's extension developing technology, Jetpack. It should be noted that this technology is obsolete now as Firefox uses WebExtensions API.

The paper specifically focuses in capability flows: add-ons elevating privileges/permissions through using code-features such as import or export. It detected leaks in four core modules that were developed by Mozilla itself and in seven Jetpack addon.

**Analysis Notes / Relevant Excerpts**

The paper specifically focuses in capability flows: add-ons elevating privileges/permissions through using code-features such as import or export. Ultimately, it detected: <<"12 capability leaks in four core modules[modules developed by Mozilla itself] and another 24 leaks in seven Jetpack addons.">>

|--> This is related to Tags #1 and #9.

The paper goes on to explain how these capability leaks take place, divided in three categories: <<"The first category of leaks occurs due to export of capabilities through direct references of privileged objects or due to function objects which return capabilities on invocation. The second class of leaks occurs when a module attaches a capability to an exported function’s this object. The third class of capability leaks occur if the module utilizes the functionality of a core module which itself leaks capabilities, such as window-utils or xpcom.">>

|--> This is related to Tag #1

<<"...the Preferences module from ‘Customizable Shortcuts’ [1], a popular Jetpack addon with over 5000 users. This module exports a method getBranch which inadvertently enables access to the browser’s entire preference tree. If another module imports the Preferences module, it would receive additional capabilities to access and modify the user’s preferences for all extensions without explicitly requiring access to the user preferences; in effect the importing module becomes over-privileged. Although the Jetpack framework recommends adherence to POLA, it does not safeguard against developer mistakes, with the result that unintended capability leaks are frequent.">>

|--> This passage in the paper explains how different modules with different permissions might end up having access to code they did not ask for permission and did not have permissions. This related to Tags #1 and Tags #9 as they discuss the access plugins have to code or objects they're not supposed to.

Tag #23

On explaining how the Mozilla addon SDK works, the paper mentions: " On execution, the Jetpack runtime loads each component module in a separate sandboxed environment resulting in namespace separation for code within the modules. Inter-module communication is facilitated by special JavaScript constructs, exports and require, which serve as well-defined entry and exit points for the modules." This is related to Tag #23 that talks about sandboxing mechanisms.

Some consequences discussed

- Privilege elevation

- Data leakage to other plug-ins

Some mitigations discussed:

- Limit plug-ins exposure to OS processes (This is related to Tag #23 that talks about sandboxing mechanisms)

- Compartmentalization of plug-ins (Tag #23's mitigation mechanism)

Hardware-software Collaboration for Secure Coexistence with Kernel Extensions (Oliveira et al, 2014)

**Summary**

This is effectively a continuation of their previous paper Oliveira et al, 2010.

The paper deals with OS kernel's extensions and security practices from untrustworthy extensions at OS and hardware level.

The paper argues that “OSes should evolve to closely interact with the hardware playing an active role in maintaining safe interactions with their extensions. Similar to the immune system, computer systems should control the interactions between extensions and the original kernel lessening the potential for security breaches. In this paper, a hardware-software (HW-SW) collaborative architecture for OS defense is proposed.”

Overlapping vulnerability types:

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

Tag #15 Breaches in Plug-ins Compartmentalizations

\* Tag #9 and #14 Improper origin check of requests by plug-ins

It also touchs upon Isolated object domains;

sandFOX: Secure Sandboxed and Isolated Environment for Firefox Browser (Saini et al, 2015)

**Tags**

Improper Isolation of Objects used in Plugins or Plug-and-play environment; Lack of a sand-boxing mechanism

**Summary**

The paper presents policies for sandboxing environments at OS level, based on SELinux, that separate the browser process from the OS, through enabling extensions to function properly in restricted environments. The limited environment restricts the access of extensions to files, to network, and to OS processes. This way the OS resources are protected from attacks related to extensions. The authors evaluated the proposed method and found that the attacks were made much more difficult and their impact was lessened after the implementation of this policy.

**Analysis Notes / Relevant Excerpts**

“To evaluate the sandFOX resilience to browser attacks, we execute several attacks on Firefox browser. We used several extensions taken from different sources [20, 23] and developed by us. These extensions are designed in such a way that they constitute the malicious functionalities of accessing critical OS resources. Table 1 illustrates some example extension along with their functionalities. Our proposed sandbox and isolation policies make many attacks considerably more difficult and lessen the impact of the exploits. In particular, we exploit major OS resources to show how sandFOX is effectively preventing compromise of these resources from an attack. Our goal, when performing this experiment is only to verify that the browser does not allow a third-party extension to compromise OS resources.”

“File System Policies: Instead of allowing the application direct access to all directories in a file system, the sandFOX allows access to tmp and home directory. Each directory is assigned its own SELinux context, making it impossible for other application to access browser information and browser application to access files in other directories. sandbox − M − H home\ −T tmp\ f irefox

Network Policies: Firefox is pretty useless without network access. Using sandbox\_web\_t allows for web browsing. sandbox − t sandbox web t −i/home/anil/.mozilla − Xf irefox The -t option in sandbox tells browser which SELinux context to use. The -i /home/anil/.mozilla tells the sandbox to copy the contents of .mozilla directory into the sandbox. This should not have any sensitive information in it such as stored passwords and cookies. The -X tells sandbox to launch an X sandbox, and the firefox is the command to run. After this command, the Firefox would no longer be able to connect to any other port except port 80, because sandbox\_web\_t is only allowed to connect to http\_port\_t

Process Policies: SELinux provides a finer-grained level of control over processes in the Linux operating system. SELinux allow to define a security policy that provides granular permissions for all processes executed and loaded by a local user or root. We use these policies in our sandFOX environment to restrict browser in handling Linux processes. Our policies enforce access controls to processes that are believed most likely to be the targets of an attack on the system. The targeted processes run in their SELinux domain, known as a confined domain, which restricts access to sensitive files such as passwd by an attacker process. If SELinux detects that a targeted process is trying to access resources outside the confined domain, it denies access to those resources and logs the denial.”

VULNERABILITIES

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

Tag #8 - Allowing a sandboxed child process to manipulate the parent process handle

CONSEQUENCES

- Data leakage

- Arbitrary code execution

- Memory corruption

- Disrupt the PnP Environment

SOLUTION

- Secure and Isolated Communication (Compartmentalization of plug-ins)

CONTEXT

- Plug-ins Execution

The Darker Side of Firefox Extension(Saini et al, 2013)

**Tags**

Improper Isolation of Objects used in Plugins or Plug-and-play environment; Improper Object Access Control in Compartmentalized PnP Environment

**Summary**

The paper focuses around the possible exploits and attacks that use extensions and the points of attack that these exploits could start off of. The paper as a proof of concept develops five extensions named: Input Grabber, Page Modifier, Overriding wrappers, Injecting BeEF(Browser Exploitation Framework) and Bypass Same Origin Policy to highlight weaknesses in Firefox PnP. This is related to our tags about improper object isolation. The paper also suggests four mitigation methods: Limit DOM Access, Integrity Checking, Least Privilege and Avoid Dangerous Methods. Three of this are related to isolating or limiting access. The last one, integrity checking, has to do with continuous checking of the integrity of the extension, especially after updates that is somehow related to our ‘elevation of privileges after updates’ tag.

**Analysis Notes / Relevant Excerpts**

It is an interesting paper that discusses all the weaknesses in the old Firefox Plugin Architecture. The overlapping concepts are:

VULNERABILITIES:

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

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

CONSEQUENCES:

Input Grabber

Page Modifier

Overriding wrappers

Injecting BeEF

Bypass Same Origin Policy

MITIGATIONS:

Limit DOM Access

Integrity Checking

Least Privilege

Avoid Dangerous Methods

Security for Middleware Extensions: Event Meta-data for Enforcing Security Policy (Shand and Rashbass, 2008)

**Analysis Notes / Relevant Excerpts**

The paper discusses the security policies between middleware features developed as extensions.

"middleware extension policy specification helps prevent unintended interactions that compromise security or reliability."

[they] "have shown how tagging message meta-data in the middleware can provide strong static and dynamic security guarantees."

The extension policies work as follows:

>> 1. Static consistency checks of the read and write permissions of each extension (at configuration time).

>> 2. Static consistency checks of the order in which extensions process an event (at configuration time).

>> 3. Dynamic checking of extension policy, where static checks are inconclusive (in the message broker).

Vulnerability types uncovered: Tag #9 - Improper origin check of requests by plug-ins

Mitigations discussed: Declarative-based request for accessing data/functionality

Vetting Browser Extensions for Security Vulnerabilities with VEX (Bandhakavi, 2011)

**Summary**

The paper presents a tool named as VEX to perform the detection of malicious flows on browser extensions. It performs a static analysis of the extension’s source code in order to detect malicious behavior. This tool is aimed to help the process of manually vetting browser extensions for untrustworthy behavior.

**Analysis Notes / Relevant Excerpts**

The paper deals with browser extensions and exploits of their vulnerabilities but it seems like it lacks an architectural approach.

Its discussion overlaps with Lack of fine-grained and modular permission setting (#malicious code being executed with elevated privileges by extensions).

Automated Security Testing of Web Widget Interactions (Bezemer et al, 2009)

**Summary**

Analyzes Ajax requests to attempt to automatically detect the following 2 vulnerabilities

"(1) a malicious widget changes the content (DOM) of another widget, and (2) a widget steals data from another widget and sends it to the server via an HTTP request'

**Analysis Notes / Relevant Excerpts**

The paper's goal is presented here: "In this paper, we propose a new dynamic analysis approach that automatically detects and reports inter-widget interaction violations." which relates to our Tag #15 (Breaches in Plug-ins Compartmentalizations).

It specifically aims to detect these violations:

V1 A malicious widget (MAL) changes the DOM subtree of another innocent widget (VIC), e.g., MAL changes the action URL of a form in VIC so that when the user submits the form, the content is sent to a malicious site.

V2 MAL reads data from VIC and sends it to a server using an HTTP request, e.g., user logs into an email account using VIC, MAL logs the entered username and password and sends it to a remote server.

WordPress Security: An Analysis Based on Publicly Available Exploits (Trunde and Weippl, 2015)

**Summary**

The paper focuses on SQL injection vulnerabilities in WordPress plugins. The paper studied 199 publicly disclosed SQL injection exploits for WordPress and its plugins. In particular they observed “the steps an attacker would take to uncover and utilize these bugs are followed in order to gain access to the underlying database through automated, dynamic vulnerability scanning with well-known, freely available tools”. The paper argued that a combination of static, dynamic and manual analysis is necessary to fully uncover SQL injection vulnerabilities.

**Analysis Notes / Relevant Excerpts**

Though the paper is framed as an empirical study of frameworks/tools/attack patterns for uncovering/exploiting SQL injection, it overlaps with a concept that we observed in our theory: Tag #19 Unsanitized plug-in data.

Impact of Plugins on the Security of Web Applications (Walden et al, 2010)

**Summary**

The paper studied vulnerabilities in 12 OSS Web applications and 13,778 of their corresponding plug-ins. The work used static analysis tools to find the vulnerabilities in those plug-ins.

They compared web vulnerabilities density in core to density in plug-in code. They found that “*While most of the vulnerabilities (92%) were found in plugin code, most of the code (93%) was also found in the plugins, and we did not find any indicator that plugin code was of lower security than core code in general. Four web applications had higher plugin vulnerability densities when compared to core code, and eight applications had lower plugin vulnerability densities*.”

**Analysis Notes / Relevant Excerpts**

The paper discusses many injection problems: Command Injection, Dynamic Code Eval, Header Manipulation, Variable Overwritten, Resource Injection, SQL Injection, Cross Site Scripting.

It also talks about Broke Authentication and Session Management: Insecure Randomness, Cookie Security, Password Management

Although the paper is more focused on comparing the security levels of plug-ins, it implicitly discusses an overlapping vulnerability type: “Tag #19 Unsanitized plug-in data”

Secure Web Browsing with the OP Web Browser (Grier et al, 2008)

**Summary**

The paper proposes a new Web browser (OP) that is mean to have “built-in” security. While the focus is a Web browser, the paper discusses some architectural decisions for supporting a plugin-based architecture with security and protection measures.

**Analysis Notes / Relevant Excerpts**

Tag#23 - Lack of a sand-boxing mechanism

In the description of the architecture of their newly created browser, the authors mention: "..(b), each subsystem runs in a separate OS-level process, and each plugin instance runs within a separate OS-level process." when talking about an individual web page instance.

The paper also mentions: "*We run each plugin object in an OS-level process and plugin objects also access DOM elements through the HTML engine.*"

Regarding communication: "*All of the processes communicate through the browser kernel, except for the HTML rendering engine and the plugins which communicate directly with a Xvnc server. The Xvnc server renders the elements locally and streams them to the UI component, via the browser kernel, where they are displayed for the user*".

Lastly, "Plugins already have welldefined interactions with the rest of the browser so we break each plugin instance into a separate OS-level process to provide the necessary level of isolation."

Tag #15 Breaches in Plug-ins Compartmentalizations

" [messages] must pass through the browser kernel, and the browser kernel implements our access control mechanism that can deny any messages that violate our access control policy."

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

"... we rely on the HTML engine to tag JavaScript code and browser plugins with the proper source domain. We use domains in our security policies to isolate different scripts and objects on the same web page; the HTML engine sets the domain and the JavaScript component and the browser kernel enforce isolation between different entities."

The number of components the plugin can talk to is limited as specified here: "The plugin freedom policy provides local plugin storage and unlimited network access at the cost of access to DOM elements and other browser components. To implement this policy, using the OP browser access controls we simply prohibit communication between the plugin and any browser

components, except the network and storage subsystem."

Tag #14: Message passing mechanism does not indicate who sent the message

In the following passage "To enforce security policy we interpose on message passing in the browser kernel. Each browser process is labeled with a security context (i.e. domain) depending on the security policy being used. We run each instance of a plugin in a separate process that is assigned its own label by the kernel. In order to correctly label each plugin process the browser kernel inspects messages that trigger the plugin to load content from a URL. This security label is then used to make decisions for other plugin and browser actions. The plugin can be denied access to browser resources; similarly, the rest of the browser can be denied access to plugin resources." it it said that the OP makes sure that the plugins are assigned their own labels during message passing and it can be inferred it will be used to check for identity.

Tag #9 - Improper principal check

To limit the access of plugins to different objects the OP does the following: "The provider domain policy sets the origin of the plugin to the site hosting the plugin content." This is done to give ".. a plugin embedded in a page permissions associated with the source of the plugin content." The paper further supports this by stating: " Label differences force separation between the content and prohibits the embedded content from altering the page or fetching any resources associated with the www.uiuc.edu domain[the page in which they're embedded]. Each of the plugins embedded inside the page can access data associated with the corresponding domains." This obviously is done to secure the core application from the plugin as stated here: "This policy limits plugins included across domains and prohibits plugin content included in this way from accessing cookies, DOM elements and other browser components."

[this might also be TAG 15]

Tag #18: Unsafe directory for plugins

OP wants to make sure that each plugin has a safe environment for its files:

"The storage subsystem provides a location for per-plugin storage that is allowed by this policy. Per-plugin storage allows plugins to have access to local settings and save files in a safe environment."

In the paper it is mentioned: "For browser plugins we use existing plugins written in unsafe languages since there are too many plugins for us to re-write them." which leads me to think that they have used a finite set of plugins.

A Practical Experience on the Impact of Plugins in Web Security (Fonseca and Vieira, 2014)

**Summary**

The paper studies the usage of static code analysis programs to detect vulnerabilities in Web apps’ plug-ins. They observed these tools effectiveness in detecting vulnerabilities and the security impacts of enabling plug-ins to extend functionality from the core.

Their results indicate a widespread occurrence of SQL injection and XSS in plug-ins.

**Analysis Notes / Relevant Excerpts**

Although the paper focused on static analysis tools and plug-ins vulnerabilities, it touched upon something we also observed it overlapped with our Tag #19 (Unsanitized plug-in data).

Analyzing Information Flow in JavaScript-Based Browser Extensions (Dhawan and Ganapathy, 2009)

**Summary**

The paper described *Sabre* (Security Architecture for Browser Extensions): “a system that uses in-browser information-flow tracking to analyze JSEs” (JSE= JavaScript-based browser extensions).

Sabre precisely tracks information flows across different browser subsystems, including the DOM, local storage and the network.

Sabre incorporates support to declassify or endorse information flows.

"Sabre, an in-browser information- flow tracker that can detect confidentiality and integrity violations in JSEs, enabled either because of malicious functionality in JSEs or because of exploitable vulnerabilities in the code of a JSE"

**Analysis Notes / Relevant Excerpts**

Tag #23 - Lack of a sandboxing mechanism

Paper discusses detecting Inadequate sandboxing of JavaScript in a JSE

Even if a JSE is not malicious, vulnerabilities in the browser and in JSEs may allow a malicious website to access and misuse the privileges of a JSE

Filtering malicious routines in web browsers using dynamic binary instrumentation (Min et al, 2012)

**Summary**

The paper presented a tool that implements a filtering mechanism to block malicious code from being executed. The filtering is implemented through a blacklist of dangerous functions and corresponding arguments of these routines.

**Analysis Notes / Relevant Excerpts**

The paper talks about Tag #23 (Lack of compartmentalization of plug-ins), as it purports a method to separate the plugin from the rest of the process. The main focus is not architecture but a filtering method to block vulnerabilities of extensions from permeating to the core application.

Here, they discussed a new concept we haven’t covered in our study: Runtime monitoring of plug-in's behavior:

\* Binary Instrumentation: Filtering is done by keeping a blacklist of dangerous routines and arguments separately to compare with the routines being called by the web browser at runtime

A Model-Based Detection of Vulnerable and Malicious Browser Extensions (Shahriar et al, 2013)

**Summary**

The paper presented a model-based technique to detect benign but vulnerable plug-sin as well as malicious plug-ins. These models are built based on features that were extracted from real widely used browser extensions. A prototype of their approach was implemented for Mozilla Firefox extension system.

**Analysis Notes / Relevant Excerpts**

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

Memo#19: Unsanitized plugin data

The paper tangentially relates to our topic as it deals with malicious extensions, but it doesn't consider the architectural side and it doesn't take an architectural approach.

Browser's defenses against reflected cross-site scripting attacks (Mewara et al, 2014)

**Summary**

The paper studied the effectiveness XSS filters implemented in Web browsers in preventing XSS exploits. Through experiments they argued that client-side techniques would be more effective in preventing XSS problems.

**Analysis Notes / Relevant Excerpts**

The focus is on cross site scripting attacks and general vulnerabilities that can happen in plugins, which overlaps with “Unsanitize plugin data”. The paper does not take an architectural approach, but it brings a mitigation that we haven’t obversed in our study: Client-side defense using XSS filters.

Enhanced browser defense for reflected Cross-Site Scripting (Mewara et al, 2014)

**Summary**

**“This paper proposes and implements an approach based on encoding unfiltered** reflections for detecting vulnerable web applications which can be exploited using above mentioned sophisticated attacks. Results prove that the proposed approach provides accurate higher detection rate of exploits. In addition to this, an implementation of blocking the execution of malicious scripts have contributed to XSS-Me: an open source Mozilla Firefox security extension that detects for reflected XSS vulnerabilities which can be considered as an effective solution if it is integrated inside the browser rather than being enforced as an extension.”

**Analysis Notes / Relevant Excerpts**

The focus is not the same as ours even though the security of plug-and-play architectures is evaluated. However, the proposed tool XSS-ME brings a mitigation technique that relies on client-side analysis of code.

CLOUBEX: A Cloud-Based Security Analysis Framework for Browser Extensions (Das and Zulkernine, 2016)

**Summary**

The work performs Static Analysis on plugins to find the following problems:

* Access Browser Information
* Change Browser Settings
* Interact with low level layers
* Interact with device sensors
* UI and Dom Related
* External Security Issues: Download Malicious Files, Send Cross Origin Requests, Modify HTPP Requests

**Analysis Notes / Relevant Excerpts**

"static analysis tasks are executed in advance in a high-speed cloud server and security analysis reports are stored in a database repository"

Performs static analysis on plugins and stores them on the server to give recommendations to users about the plugin they are installing

This seems like less of an architectural approach and more scanning extensions for “Unsanitized plug-in data” vulnerabilities.

Browshing a new way of phishing using a malicious browser extension (Varshney et al, 2017)

**Summary**

The paper investigated a technique that attackers can use to perform phish attacks through malicious browser extensions. They demonstrated the technique through a Chrome extensions.

**Analysis Notes / Relevant Excerpts**

Tag #2: Incorrect user notification of plug-in permissions? (More addresses using permissions in a way that confuses the user because the user does not understand what can be done with each permission)

Flaws in browser design and security policies which allows the browser extension to access sensitive data through a vague permission based architecture.

No static or dynamic detection techniques are capable of detecting the zero day attacks such as Browshing and many more via malicious browser extensions.

Primary purpose of the paper is to make it known that browser extensions could be used for phishing. Does not take an architectural approach.

Detecting DOM-Sourced Cross-Site Scripting in Browser Extensions (Pan and Mao, 2017)

**Summary**

The paper proposes a hybrid framework to detect DOM-based XSS attacks. The framework performs a static analysis to find potential vulnerable candidate attack surgaces. Then, it employs a dynamic symbolic execution to generate a document (DOM) as a proof-of-concept exploit.

**Analysis Notes / Relevant Excerpts**

The paper tangentially touches upon the subject as it deals with vulnerabilities in browser extensions but it does not have an architectural approach. In particular focused on XSS vulnerabilities via DOM manipulation. However, it overlaps the problem of unsanitized plug-in data.

PyXhon: Dynamic detection of security vulnerabilities in Python extensions (Sun et al, 2012)

**Summary**

The paper presented PyXhon. The primary goal of PyXhon is to evaluate third-party extensions by monitoring the function-call procedures and analyzing byte instructions traces so as to figure out the whole malicious behaviors and potential security risks.

**Analysis Notes / Relevant Excerpts**

Function oriented analysis to monitor the types of operations being performed by the plugins

"The primary goal of PyXhon is to evaluate third-party extensions by monitoring the function-call procedures and analyzing byte instructions traces so as to figure out the whole malicious behaviors and potential security risks."

"If developers want to verify whether the Python code has accessed system files, it is convenient to monitor the built-in function execfile(), open() and file(), because Python developers will not overwrite those internal key functions."

"Our strategies are totally different from the sandbox security evaluation approach and restricted execution mode, but to provide the independent runtime environment for third-party extensions"

These overlaps with Lack of compartmentalization plug-ins.

Mitigating Drive-by Download Attacks: Challenges and Open Problems (Egele et al, 2009)

**Summary**

The paper proposed techniques to mitigate the problem of drive-by downloads trieggered by malicious Web sites.

**Analysis Notes / Relevant Excerpts**

**“**we argue that browser vendors need to integrate mechanisms into their browsers to efficiently protect Internet users against drive-by download attacks.”

“For the shortcomings mentioned above, we envision protection mechanisms built into the browser itself. As the decision whether a page is malicious or not is reached during the download and interpretation of the page itself, such techniques do not suffer from outdated information. Furthermore, performing the analysis only on the currently visited page does not create additional load for other non-related sites. As many drive-by attacks rely on client-side scripting, we focus on mechanisms that allow for detection of such attacks.

The information required to cast a decision on the maliciousness of a page can be gathered from scripts in two ways. Either statically by analyzing the page and the scripts it contains, or dynamically, where the analysis is performed during the execution of the scripts. The main advantage of static analysis over dynamic analysis is that all possible execution paths can be taken into account. If a script, for example, exhibits malicious behavior only if viewed with a specific browser, static analysis could still reveal the malicious intents of the script, even when visited with a different browser. Obfuscation and encryption schemes, on the other hand, give an attacker an easy means to prevent efficient static analysis.”

These discussions overlap with the problem we also found “Bypassing user notification or restriction checking process for plug-in installation”. It also proposes two techniques that we have not observed in our theory: Anomaly based detection method and Emulation-based mitigation technique.

“Emulation-Based Mitigation Technique: Drive-by attacks also make use of exploits that rely on shellcode. We envision that these threats can be mitigated by applying emulation techniques similar to approaches that are used to detect shellcodes in network streams [18]. To this end, we propose an approach where the memory contents that contain data retrieved from a web site are examined for the longest valid instruction sequence of machine instructions [7].”

Static and Machine Learning Approaches: To prevent malicious scripts from using APIs in unintended ways, we propose a technique that infers sets of possible values or domains of parameters.”

Title (et al,)

**Summary**

It discuses extensible architectures and the difference between traditional and pure plugin architectures, which are systems whose entire functionality comes from plugins. This paper also points out specific areas to be careful of when defining a plugin architecture. Those areas are installations, updates, configuration, discoverability, and security. This work uses Eclipse as an example of a plugin architecture and briefly discusses decisions made by eclipse.

**Analysis Notes / Relevant Excerpts**

The paper nicely describes some challenges when designing a plugin architecture, which are:

“Installing and updating. Many modern products automatically detect when they are out of date with respect to available service or product version. Either on start-up, or as a result of an explicit update action, the products compare the current installation level against some network-based baseline. The product then automatically downloads required fixes or upgrades and applies them, often as part of the product execution. Additionally, users can, and most often will, install additional plug-ins from various sources to extend the functionality provided by their current application.”

“Security. Systems can never be too secure, and you need to pay particular attention to securing a system based on plug-ins. Since arbitrary plug-ins can be installed—for example, by downloading them from the Web—and are allowed unlimited access to the system they plug into, security in a plug-in environment must be carefully planned. On top of this, some plug-ins require support for executing custom install code during installation, so they can have control over some parts of their installation. To prevent software security accidents or failures, the plug-in framework must address the issues of downloading from third parties and controlling a plug-in’s access to other code and data”.

“Concurrent plug-in version support. Without a doubt, managing concurrent plug-in versions and dependencies is one of those problems that can keep architects, developers, and installation folks awake at night. Most of you have probably experienced “DLL hell” at some point and will look with suspicion on something that has the potential of being a “plug-in hell.””

“Scalability, up and down. Another challenge of working with plug-in architectures is scalability. Just as the problems associated with multiple versions and dependencies can quickly escalate, so too can the sheer number of interacting plug-ins quickly become a problem. For example, when Eclipse was first designed, it was thought that a product, a large one for that matter, would consist of a few hundred plug-ins. A few releases later, some enterprise-class products built on Eclipse are known to have passed the thousand plug-in mark, so the platform goal has been revisited to support scaling between 5,000 and10,000 plug-ins.”

“Obviously, the new plug-ins, private to that user, cannot be installed in the read-only, shared install location, so the product should allow users to install and configure extra plug-ins in a location where they have more privileges.”

These overlap with the vulnerability type:

“Extraction/storage of plug-in with world-readable/writable permissions or in unsafe directories” and having a “Dedicated secure storage” to mitigate it.

Fast and Vulnerable: A Story of Telematic Failures (Foster et al, 2015)

**Summary**

The paper demonstrated all the technical details of using vehicular “dongles” (Telematics Control Unit – TCU) to exploit a car. These TCUs can be plugged into the car via an OBD-II port type. This port has a series of protocols that allows these dongles to access an array of low-level sensor data about the car’s operation. Moreoevr, the OBD port also provides access to the CAN bus

**Analysis Notes / Relevant Excerpts**

“In spite of the fact that most aftermarket TCUs are designed for monitoring only, CAN is a multi-master bus and thus any device with a CAN transceiver is able to send messages as well as receive. This presents a key security problem since as we, and others, have shown, transmit access to the CAN bus is frequently sufficient to obtain arbitrary control over all key vehicular systems (including throttle and brakes).

Further, it is common that these devices, in addition to a microprocessor and a CAN transceiver, also provide a number of additional functionality including a GPS, accelerometers and, critically, external networking connectivity. This latter feature may be as simple as a short-distance Bluetooth interface (e.g., as with Automatic Lab’s dongle), but more commonly is a full cellular modem (2G or 3G) that provides remote data connectivity via the Internet. Taken together, these pieces of functionality place tremendous weight on the security of aftermarket TCU software. Should such software be vulnerable to external compromise, this would allow an attacker to control a wide array of vehicles at arbitrary distance.”

“The CAN bus standard was designed by the auto industry to provide a way for the various electronic control units (ECU) in an automobile to communicate. It forms a simple bus topology network in which messages are tagged with identifiers that are associated with particular ECUs or functions (the layout of a sample CAN frame is shown in Figure 1). However, there is no enforced addressing or message source identification and thus any device on the CAN network can send any message in a manner indistinguishable to the target ECU..

Their discussions along the paper focused on the vulnerabilities in the dongle itself (plug-in). However, these discussions also implies a Lack of compartmentalization of plug-ins and Lack of fine-grained and modular permission setting.

A Safety Aware Run-Time Environment for Adaptive Automotive Control Systems (Frtunikj et al, 2014)

**Summary**

The paper changes to the electrics and electronics architecture of cars. It proposes a Runtime Environment (RTE) that can better orchestrate the interaction among components and ensure safety.

**Analysis Notes / Relevant Excerpts**

“The Platform Supervision service handles hardware (random) and software (systematic) faults, as well as illegal access in combination with the Security Manager. Its main functions are: reception of MonitoringIndication messages from different components, aggregation and inference of these indications to produce ConsolidatedIndications (Figure 2). The latter set is then synchronized with other nodes in order to achieve a consistent view on the system state as a whole. ConsolidatedIndications are then updated and made available to the state management components. Systematic errors in specific functions are signalled to the Application State Manager service, which performs necessary recovery of the function. Detected faults currently classified as random hardware faults or RTE systematic faults are passed to the Platform Mode Manager and also accumulated locally to achieve statistically sound diagnosis of intermittent faults. The process of fault detection, consolidation and reaction and the corresponding mapping to RTE components is depicted in Figure 3.”

Although the paper is focused on a hardware architecture, it touches upon concepts of software running in these embedded systems. In particular, it overlaps with Reentrant event callbacks.

Detecting and Analyzing Insecure Component Usage (Kwon and Su, 2012)

**Summary**

The paper presented a framework for detecting and analyzing vulnerabilities that arise from the insecure usage of components in a software. The framework attempts to detect inconsistent security policy configurations. The component usage is modeled in terms of memory usage, and conditional jumps that changes the program counter.

**Analysis Notes / Relevant Excerpts**

“To detect inconsistent policy configurations formalized in Definitions 2.6 and 2.7, it is necessary to analyze how security policies are set and enforced in code. However, it is challenging, because 1) most components are distributed as binaries, and 2) it is difficult to know which memory locations are used for security policy configuration and evaluation. To address this issue, at the high-level, we design our framework as two phases: runtime extraction and offline detection. In the runtime extraction phase, we instrument executions of a reference and a test subject to capture security policy-related executions (Definition 2.4). We perform an offline analysis to detect insecure component usage in the captured executions. Although the high-level approach appears straightforward, the main challenge is how to perform scalable and precise analysis. For example, IE performs millions of memory accesses at runtime, and it is practically infeasible to instrument and analyze all of them. To address this scalability issue, our framework uses the following optimizations: 1) instrumenting target component execution, 2) filtering irrelevant memory accesses, and 3) performing preliminary analysis on policy evaluation.” Although the paper focused on Windows application, its discussions brought concepts from Improper object access control in compartmentalized PnP environment.