I. Results of the Study

We conducted an in-depth study of seven systems with plug-and-play architectures. In total, we have analyzed 3,183 vulnerabilities from Chromium, Thunderbird, Firefox, Pidgin, WordPress, Apache OfBiz, and OpenMRS whose core architecture is based on a plug-and-play approach. We identified 15 common types of security problems that occur in plug-and-play (PnP) systems. In the following subsections, we describe all the results within their identified context (e.g. PnP functional context).

A. Plug-in Install

One of the most basic features in a plug-and-play system is to *load and install new plug-ins* to the application at runtime. In this context, we found the following types of problems:

1) Incorrect user notification of plug-in permissions: When a new plug-in is added to the system, it can request access to certain data/functionality provided by the PnP environment. This problem occurs when the PnP environment does not (or incorrectly) shows the list of data and/or functionality that will be accessed by the plug-in before the plug-in is installed.

– *Mitigation*: Similar to the vulnerability "Bypassing user confirmation for plug-in installation" (Section I-A2) the mitigation is to adopt a *Central Install Point* that would display a list of all the permissions being requested by the plug-in (besides consistently generating for mitigating the problem discussed in Section I-A2).

2) Bypassing user confirmation for plug-in installation: Whenever a new install is requested, the PnP environment should ask the user for consent to proceed (or abort) the install. This type of vulnerability is caused by not strictly enforcing having all install requests mediated by the user.

– *Mitigation*: It consists of having a *Central Install Point* in which all the installation requests, regardless of how initiated, are guaranteed to go through this component. This component ensures that the install process of a plug-in is designed to prompt the user for consent before proceeding with the install.

3) Lack of plug-in's configuration file sanitization: This problem is caused by not validating the plug-in's configuration file in order to verify whether it is structurally correct, and also escape/neutralize any code that is injected in the plug-in's configuration file.

– *Mitigation*: Validating mechanisms must be established and applied before the plug-in is added to the registry. For instance, the PnP environment can define a typed data structure that specifies the expected data type in each field within the configuration file. Each raw field in the configuration file is converted to the corresponding field in the typed data structure. This typed data structure is validated, in which a validator properly escapes and neutralizes any code injected in one of the configuration file's fields. Then, if the validation was successful, this typed data structure is passed to the initialization routine that will perform the necessary tasks for adding the plug-in to the plug-in's registry.

4) Improperly checking the origin of an install request: This vulnerability occurs when the PnP environment accepts install requests initiated either by the user or an external entity (i.e., a remote install), but it does not check (or incorrectly checks) the source of an install request.

– *Mitigation*: It can be fixed through defining a *whitelist* of the trusted remote sources that are allowed to trigger an install.

B. Plug-in Updates

1) Elevation of privilege through a plug-in update: This vulnerability occurs when plug-ins specify a list of privileges upon install and the user accepts these permissions. However, the PnP environment does not check for the changes in privileges of plug-ins after an update, bypassing the user consent. Therefore, a plug-in can elevate its permissions through a plug-in update, and without user consent.

– *Mitigation*: Permissions need to be enforced at lifetime: once a user confirms a set of permissions during install, these cannot be elevated at any period of time.

C. Plug-in Registry Management

1) Extraction/Storage of Plug-in with worldreadable/writable permissions or in unsafe directories: In general, plug-ins are released as software bundles (e.g., zip files) that are extracted by the PnP environment. When the PnP environment extracts and/or stores these bundles using world-readable (or writable) permissions (e.g. 777 permissions in Unix-based operating systems), any other plug-in or potentially external process can alter plug-ins' data or code.

– *Mitigation*: Each plug-in's bundles (configurations, scripts, binaries and other related artifacts) must be stored in a read-only storage dedicated to that plug-in.

D. Plug-and-Play Execution Environment

1) Lack of compartmentalization of plug-ins: This vulnerability type is caused by a lack of a well-defined logical compartment to isolate plug-ins from each other and from the PnP environment.

– *Mitigation*: There are two complementary ways of fixing this vulnerability. The first one is to create logical compartments and make operations outside that compartment limited and intermediated by the PnP environment. The second approach is to create isolated object domains , as previously discussed in Section I-D7.

2) Lack of fine-grained and modular permission setting: Many vulnerabilities observed in our analysis were due to benign plug-ins that had more privileges than needed to implement their features. A fine-grained and modular permission setting could have limited the access of such plug-ins.

– *Mitigation*: is to modularize the PnP environment into different fine-grained related functions with specific privileges. Plug-ins will gain access to specific functions not all.

3) 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: The vulnerability arises from the scenario in which a plug-in, executing in a unprivileged process, tampers with a high-privileged process in order to escape its security boundaries.

– *Mitigation*: This problem can be mitigated by limiting plug-ins' exposure to high privilege PnP APIs. Furthermore, OS system calls to other processes running in the underlying operating systems must be prevented. This means leveraging a mechanism that intermediates any system call between the plug-in and the underlying OS. The second mitigation technique is to limit the access to higher-privileged APIs.

4) Improper object access control and compartmentalization enforcement: When plug-ins are isolated in different logical compartments, they communicate with each other through object proxies that enforce compartment's access policy. Security issues can occur when the PnP environment uses an incorrect proxy for the inter-compartments communication.

– *Mitigation*: The PnP environment *enforces security policies through object wrappers*, which act as proxies for a real object residing in a different compartment. These wrappers are instantiated and used according to the relationship between the caller and the callee compartments. Each type of object wrapper enforces a different type of security policy, which indicates the properties and operations would get accessed by the callee compartment.

5) Unsanitized plugin data: The core application interacts with data from the plug-ins. Security problems arise when the PnP environment trusts data from the plug-in and, therefore, it does not properly sanitize the data.

– *Mitigation*: Adoption of an input validation mechanism that intercepts and sanitizes the data flowing from plug-ins to the core application.

6) Improper origin check of requests by plug-ins: can result in a security breach when the PnP environment fails to correctly check the origin of requests (i.e., who was the plug-in that initiated a call), therefore allowing the elevation of privilege attack.

– *Mitigation*: Each plug-in must be assigned a unique identifier that acts as an origin identifier. Then, the PnP environment must check the origin of requests against a security policy whenever a new incoming request is made to the PnP execution environment.

7) Improper isolation of objects used by plug-ins in the PnP environment: Plug-ins attach to the PnP environment through well-defined public interfaces/APIs provided by the PnP environment. The interaction of Plug-ins and PnP environment is through these APIs. Security problems can occur when plug-ins and the PnP environment share the same objects or data structures of these APIs. As a results, plug-ins can interfere with the PnP environment or other plug-ins.

– *Mitigation*: The PnP environment can implement an *isolated object domain* solution. Each plug-in (and so the PnP environment) must have its own copies of objects that are passed to or returned by an API call. Then, the PnP environment manages these objects, ensuring that pointers or "object references" used by a plug-in, are not pointing to objects in another compartment (PnP environment or other plug-ins).

E. Plug-ins Request Handling

1) Reentrant event callbacks: Plug-ins can interrupt the execution of the event dispatching mechanism before it has finished, resulting in an unpredictable state. 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 PnP environment perform these operations in an atomic fashion.

– *Mitigation*: Dispatch of the event to plug-ins must be atomic such that it guarantees the integrity of the PnP environment (avoid to leave the PnP environment in an invalid state).

2) Plug-ins requests are handled without authorizing plug-ins that initiate the request: Vulnerabilities arise when the plug-and-play environment accepts any call from plug-ins without checking whether the plug-in is authorized to make such API call.

– *Mitigation*: There are various mitigation techniques. First, upon a request PnP environment must authorize the plugin that initiates a requiest or subscribes to an event. Second, events must be decomposed into sensitive and non-sensitive events. Listeners can only subscribe to sensitive events if and only if they have enough permissions. Third, events and APIs specific to PnP environment must be hidden from plug-ins.