Privilege Separation

Scribe : Rohan Bavishi

September 13, 2017

1 Introduction

1.1 Principle of Least Privilege

The principle of least privilege [3] states that a particular job/process should have the minimal number of security privileges that is enough for it to retain the desired functionality. For example, a program invocation like `ls $HOME` only needs read permission for the home directory of the current user. If it were given root privileges as well and if a bug was found in the "ls" source code, it could potentially be exploited to gain root access to the entire system.

1.2 Privilege Separation

Privilege separation [2] is a technique to harden a system against possible attacks using vulnerabilities in the software, wherein the system is divided into multiple smaller components, each assigned its own set of privileges according to the principle of least privilege. Therefore, if any one component is compromised, collateral damage is limited i.e. other components are not compromised directly.

Typically, a few components will have higher privileges in order to access sensitive data, such as the password database, while other components will communicate with these components if they want to execute sensitive operations.

It is generally the case in software that the number of security vulnerabilities increases as the number of LOC in a program increases. Therefore, a monolithic structure would allow any single vulnerability to compromise the entire program. Separation into smaller components effectively reduces the size and also makes security auditing easier, as auditors can now potentially focus on the sensitive components, which are now much smaller in size thanks to the separation of the source into distinct parts.

1.3 What can Privilege Separation protect against?

Privilege separation does not prevent an exploit, it mitigates the damage caused by it. So if a relatively less sensitive component gets compomised, the attacker won't be able to gain access to more sensitive data.

However, this also implies that the technique does not prevent any Denial-of-Service (DoS) attacks. An attacker can crash some component and hinder the normal execution of the program.

2 Privilege Separation in Chromium

The reading [1] describes the architecture of Chromium that incorporates the technique of privilege separation. The architecture has the following design decisions -

- Chromium has two separate major components - the Rendering Engine (RE) and the Browser Kernel (BK).
- The Rendering Engine is basically the web - it is in charge of parsing, interpreting JS, rendering the website image etc. A separate RE process is created for each tab - this is done mainly to provide fault tolerance. Each process is run in a sandbox and is assigned separate protection domains based on privileges.
- The Rendering Engine alone is responsible for implementing the Same-Origin Policy.
- The Browser Kernel is reponsible for managing the various RE processes, and for regulating RE's access to the user's filesystem. It also manages the sensitive persistent data such as cookies, passwords, bookmarks etc. It provides an API for the rendering engine to interact with the OS and maintains security policies for each RE process. It also regulates networking for the rendering engine, in that all URL requests are serviced by the browser kernel. This is mainly to protect against urls with the `file://` scheme.

The architecture guarantees that any vulnerability in the rendering engine cannot be exploited by attackers to access the user's filesystem. However it doesn't protect against bugs in the rendering engine, especially in the implementation of the same origin policy, which can then be used to attack other websites.

2.1 Why only protect users and not other websites?

Let's look at a possible solution to protect websites as well -

- We can have a separate renderer instance for each origin (iframe origins included). We can sandbox each of them.
- The Browser Kernel will now have security policies for the renderer instances for each origin (instead of just one tab).
- Redirects need a fresh renderer instance.
- We can have javascript from different origins (and therefore different renderer instances) to interact with the HTML DOM via an API of sorts.
- ...

It is easy to see that the complexity has vastly increased, and there may still be corner cases that are missed. Performance is very likely to take a hit. Increased complexity comes with increased probability of security bugs as well.

Another very important factor is compatibility with existing websites. The solution given above may make some websites incompatible with Chromium because of the javascript being executed in separate instances and the consequent use of the communication API. If a browser is unable to display even a few websites which other browsers can, it will lose the market share fairly quickly. Therefore the designers of Chromium decided to protect against the few of the most severe of vulnerabilities (arbitrary code execution on the user's system), while still retaining compatibility with existing websites.

References

[1] A. Barth and C. Jackson. The security architecture of the chromium browser. 10 2017.

[2] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[3] J. Saltzer and M. Schroeder. The protection of information in computer systems. 63:1278 – 1308, 10 1975.