CS261 **Fall 2008**

Lecture 7: September 18

Sandboxing Part 1: A Secured Environment for Untrusted Helper Application

Lecturer: David Wagner *Scribe: Susmit Jha*

1 Introduction

This lecture was mainly a discussion on the paper *A Secured Environment for Untrusted Helper Application* [1] by Goldberg et al with focus on understanding the problems with Janus approach and issues that Janus did not resolve.

2 Overview

The paper under discussion proposed a framework called Janus to ensure that the execution of untrusted application is sandboxed. It is an user-mode tool which functions by conceptually placing itself as a reference monitor between the applications and the OS. It intercepts all system calls made by untrusted application and enforces its security policy by denying calls which violate security specification. In its implemented version, it runs as a separate process which uses tracing mechanism (such as ptrace) of the OS to intercept system calls.

<u>Conceptual view</u>

Untrusted Application ⟶ Janus ⟶ Syscall handler

————User Space———— | ——— Kernel Space ———

<u>Actual Implementation</u>

```
Untrusted Application ⟶ |
                        | Syscall Handler
      Janus        ⟵⟶ |

———User Space——— | ——— Kernel Space ———
```

3 Motivation

Many applications such as network daemons run with much more privileges than they require for their normal functioning. Consequently, subverting an application like sendmail can lead to damage to the system beyond the sendmail program. Ideally, one would like to contain an attack to within an application. This can be achieved by keeping the application processes outside one's trusted computing base and having each application process in a sandbox. The goal is to ensure that even if an application becomes malicious, it won't be able to harm rest of the system and the damage will be restricted to loss of functionality of the infected application.

4 Issues with Janus

4.1 Creating Policies is hard

There are a number of ways to address this -

- Learn policy with user interaction (many firewalls use this) - Problem is that one bad decision spoils the entire policy.
- Use division into roles and a separate policy for each role of the user - This assumes that user would be cautious in switching roles as required.
- Static Analysis - This is computationally very expensive and not a practical approach for even moderate sized applications.
- Runtime analysis - This is a commonly used technique. The assumption made is that the training phase is clean and there are no malicious activities being performed by application while learning the policy from logs.

4.2 Composition of policies and processes

If we have a policy that we will not allow any process reading from */home* to communicate over network, two processes working in collusion can read the data from a text file from */home* and transmit it over network without violating the policy. Thus, it is not only enough to have a policy for each application but it is also required to frame a communication policy between the applications.

4.3 Symbolic Links

Let us have a policy that anything in */tmp* would be allowed to be read. It is possible to then create a symbolic link $/tmp/foo$ which would point to */etc/passwd.* Now reading from this link would mean reading from */etc/passwd.*

4.4 TOCTTOU for Symlinks

Supposing we did check what the links were pointing to and did not permit opening links to point to outside temp.

One could then create a file */tmp/etc/passwd* and make a link $/tmp/x/y/foo$ which would point to *../../etc/passwd* which will pass the test since this file is in the temp directory. Now, if someone does a move from *mv/tmp/x/y/tmp*, we will have the new link $/tmp/y/foo$ which would point to *../../etc/passwd* which is actually */etc/passwd* and hence, someone can now use this link to read the restricted file.

4.5 TOCTTOU for file system

This is a rather sophisticated attack. The number of symlinks in a chain reference is restricted to some small number like 16 or 40. Each link can point to some file, the path of which is limited by some string size. Thus, one can make a symbolic link point to a max length (4096) string path such that the target is another link and so on to make a daisy chain of upto the max allowed number of links. When this dereference will be resolved, there will probably be a page fault while reading the entire path. Now, the malicious application can change the link to point to some restricted location so that when the open command is executed, the restricted location is opened. This window of attack is quite significant as this involves filesystem IO.

4.6 Not Portable

The implementation of Janus was not easily portable. There were also other issues like the overhead of *ptrace* and how to ensure that a process is killed if the tracer of that process dies for any reason. Forking was also a problem and had to be dealt in ad hoc manner.

4.7 Vulnerable to OS bugs

A very good example of this is the bug related to null pointer dereference in the kernel. A malicious application could exploit this to escape out of Janus sandbox.

Let there be some kernel code (possible related to some device driver) where we have $p \rightarrow op \rightarrow send\ (arguments)$. Supposing the application uses *mmap* to map this to NULL page and writes malicious code at some offset x where x is the offset of *send*. Now, the kernel would just inherit this configuration and run the malicious code. This arises due to the fact that NULL is in user address space and it is not possible to remove it from user address space as it would break a lot of existing applications.

4.8 Static Policy

In general, one would like a dynamic policy which depends on user's interaction as well as system state instead of a static policy. For example, a text editor should be allowed to read or write to any file that an user asks it to open to (provided the user has permissions to read/write on the file). Having a policy where a text editor can only read or write to */tmp* would be too restrictive to be of practical use.

4.9 TOCTTOU for syscall parameters

One could think of two ways to get rid of this -

- Once a process issues a system call, put it to sleep so that it can't modify any of the arguments. The problem with this is that a process can be awakened by some other malicious process or signal handlers.

- Another approach is to map the memory region with system parameters and make it read only. After system call is executed, the region is again made writable. This has heavy performance overhead.

5 Conclusion

In essence, this paper proposes two new techniques for sandboxing - virtualization and interposition. Sandboxing can be achieved by virtualizing the OS syscall interface and interposing a security enforcement agent as a layer between OS and the application program.

References

[1] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, Berkeley, CA, USA, 1996. USENIX Association.