A Software Composition Flaw in Google Desktop Search

Oren Dobzinski and Jeannette M. Wing
Electrical and Computer Engineering Department and Computer Science Department
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh PA
{orend, wing}@cs.cmu.edu

ABSTRACT

Modern software systems are composed of different modules and objects that interact with each other. Each of these components may satisfy a local security policy. It may also satisfy a global security policy with respect to its intended operating environment. However, when many components are put together, because of unexpected interactions among them, a local security policy and/or the global security policy may be violated. A *composition flaw* is when the execution of a composition of separately secure components leads to a system state in which a local or the global security policy is invalidated[1]. We are particularly interested in composition flaws at the design, not code level and therefore are currently exploring the nature of these flaws so we can detect them automatically before the composition is performed. Our long-term goal is to identify new kinds of composition flaws before attackers discover and exploit them. As a first step towards this goal we show an analysis of a recent composition flaw discovered in the Google Desktop Search application, a flaw that compromises users' privacy. We show the principles of this type of flaws and describe our approach to detecting them.

1. INTRODUCTION

Our research direction was inspired by a composition flaw that was recently discovered in the Google Desktop Search (*gds*) application, in the composition of *gds* and a Java applet (see [1]). We will briefly describe this flaw and then give a formal model for it. One of the components we describe, *gds* is a tool that allows users to search their local machines. A unique feature of *gds* is an integration of a short summary of local search results inside a regular search results page. This summary includes 30-40 character snippets of local files that contain the queried term. This integration is done by *gds* by observing all outgoing connections on the localhost. Once a google.com request is detected, *gds* performs a local search. When the results from google.com return, the local search results are integrated with the returned html page. The integrated page is returned to the initiating entity. The other component in our system is a Java applet, whose security policy states that it cannot read any local files.

[1]Notice that we can have a composition flaw even if each component is implemented correctly with respect to its behavioral specification.

While each of these components obeys the global security policy, their composition creates a flaw that can be exploited in the following way. A *gds* user visits a malicious website that contains a Java applet, which is loaded to the user's host memory. The applet connects to the attacker's host, which serves as a web proxy and performs a google search. The outgoing google query is detected by *gds*, and a local search is initiated by *gds*. The attacker's host returns a results page, possibly an old one that it cached, and *gds* integrates the search results and returns it to the applet. The search results are transmitted to the attacker's host, which can observe the snippets from localhost's files. In fact, it can initiate any search it desires with any keyword and therefore read sensitive parts of files on localhost.

2. FORMAL MODEL

Let *JavaApplets* be the set of all Java applets. Let *Hosts* be the set of all hosts. Let *Files* be the set of all files on all hosts. Let *Applications* be the set of all applications. We define *readableFiles(host, X)* $\subseteq$ *Files* as the set of files residing in *host* that may be partially or fully readable by an applet, application or a host X, where *host*$\in$ *Hosts*. Formally: Let E be the set *Hosts* $\cup$ *Applications* $\cup$ *JavaApplets*. We define a function *readableFiles: Hosts* $\times$ *E* $\rightarrow 2^{Files}$. We now fix a host *localhost* and omit *host* when we use *readableFiles* below.

Let *connect* be a binary relation, *connect* : $E \times E \longrightarrow$ *Boolean*, where *connect*(e_1, e_2) indicates that there is a flow of data from e_1 to e_2. Note that *connect* is not symmetric. It is given that *connect*(X, $host_1$) $\Rightarrow$ *readableFiles(X)* $\subseteq$ *readableFiles*($host_1$), meaning that when an applet or an application is connected to a certain host, the set of files that are readable by it is now readable by the host as well.

The global security policy we would like to maintain is: $\forall$ *remoteHost* $\in$ *Hosts*$\setminus\{$*localhost*$\}$, *readableFiles(remoteHost)* $= \emptyset$. We will show how this property does not hold in the case of the Google Desktop Search exploit.

2.1 The Java Applet Component

The first component we consider is a Java Applet, $\mathit{applet} \in \mathit{JavaApplets}$. The security policy of this component is: $\forall\ \mathit{applet} \in \mathit{JavaApplets},\ \mathit{readableFiles(applet)} = \emptyset$. In order to satisfy this policy the following property is always enforced: Let *applet.originatingHost* be the host from which the applet was downloaded. It is always true that $\forall\ \mathit{applet} \in \mathit{JavaApplets},\ \forall\ \mathit{host}_1 \in \mathit{Hosts},\ \mathit{connect(applet,\ host_1)} = \mathit{true} \Rightarrow \mathit{applet.originatingHost} = \mathit{host}_1$. This property specifies that the applet is allowed to make network connections only to the originating host of the applet. Also, for some $\mathit{applet} \in \mathit{JavaApplets}$ we denote *applet.host* as the host on which the applet is running, where $\mathit{host} \in \mathit{Hosts}$.

2.2 The Google Desktop Search Component

The second component we consider is the Google Desktop Search application, denoted as $\mathit{gds} \in \mathit{Applications}$. We denote *gds.host* as the host on which *gds* is running, where $\mathit{host} \in \mathit{Hosts}$. One property of *gds* that is always enforced is the following: $\forall\ \mathit{host}_1 \in \mathit{Hosts},\ \mathit{connect(host_1,\ gds)} = \mathit{true} \Rightarrow \mathit{gds.host} = \mathit{localhost}$, which means that the local web server run by *gds* only accepts connections to localhost (127.0.0.1). We assume that the *gds* application has access to some files on the host it runs on (defined by the user). Formally, we assume that $\mathit{readableFiles(gds)} \neq \emptyset$. Note that this component does not have a security policy. Next we describe the attack.

3. THE ATTACK

1. The user who uses the *gds* application ($\mathit{gds.host} = \mathit{localhost}$) visits the attacker webpage, which is located on the attacker's host, $\mathit{attackerHost} \in \mathit{Hosts}$. A Java applet, $\mathit{applet} \in \mathit{JavaApplets}$ is loaded to the user's host memory. Thus $\mathit{applet.originatingHost} = \mathit{attackerHost}$.

2. The applet connects to the attacker's host, which serves as web proxy and performs a google search.

3. The outgoing google query is detected by *gds*, a local search is initiated by *gds*, after the following connection is established: *connect(localhost, gds)*. Note that $\mathit{connect(localhost,\ gds)} = \mathit{true}$ since $\mathit{gds.host} = \mathit{localhost}$.

4. The attacker's host returns some results page, possibly an old one that it cached, and *gds* integrates the search results and returns it to the applet: *connect(gds, applet)*.

5. The following connection is established: $\mathit{connect(applet,\ attackerHost)} = \mathit{true}$. Such connection is allowed since $\mathit{applet.originatingHost} = \mathit{attackerHost}$. The search results are transmitted to the attacker's host, which can observe the snippets from localhost's files.

Therefore we have:

1. $\mathit{connect(applet,\ attackerHost)} \Rightarrow \mathit{readableFiles(applet)} \subseteq \mathit{readableFiles(attackerHost)}$

2. $\mathit{connect(gds,\ applet)} \Rightarrow \mathit{readableFiles(gds)} \subseteq \mathit{readableFiles(applet)}$

3. $\mathit{connect(localhost,\ gds)} \Rightarrow \mathit{readableFiles(localhost)} \subseteq \mathit{readableFiles(gds)}$

All three *connects* are possible and hold true, so $\mathit{readableFiles(localhost)} \subseteq \mathit{readableFiles(attackerHost)}$. Therefore we can conclude that $\mathit{readableFiles(localhost)} \neq \emptyset \Rightarrow \mathit{readableFiles(attackerHost)} \neq \emptyset$, since we assumed that $\mathit{readableFiles(localhost)} \neq \emptyset$.

4. THE FIX

Rather than inserting the local search results directly into the Google search result, *gds* could insert some HTML that creates an internal frame (IFRAME) element which loads its content from the *gds*s internal web server. This IFRAME would have a different source than the web page that surrounds it, meaning that hostile applets, would be unable to read the local search results. This was indeed the fix that Google chose to implement. Formally, the fix to this flaw is the following. We refine the connection conditions of *gds* by adding another property to *gds*: $\forall\ e \in \mathit{Applications},\ \mathit{connect(gds,\ e)} = \mathit{true} \Leftrightarrow e \in \mathit{IFrames}$, where $\mathit{IFrames} \subseteq \mathit{Applications}$. The new component we have is an *IFrame*, which has a property similar to the Java applet property : $\forall\ \mathit{iframe} \in \mathit{IFrames},\ \forall\ e \in E,\ \mathit{connect(iframe,\ e)} = \mathit{true} \Leftrightarrow \mathit{iframe.originatingHost} = e$. Hence, the step *connect(gds, applet)* in the attack is not allowed. We have an intermediate step *connect(iframe, applet)*, which does not happen since $\mathit{applet.originatingHost} \neq \mathit{iframe.originatingHost}$. Thus, the chain above breaks. Formally, the attack

1. *connect(applet, attackerHost)* 2. *connect(iFrame,applet)* 3. *connect(gds, iFrame)* 4. *connect(localhost, gds)*, where $\mathit{iFrame} \in \mathit{IFrames}$ is not possible since $\mathit{connect(iFrame,\ applet)} = \mathit{false}$.

5. MODEL CHECKING APPROACH

We are currently investigating a model checking approach to automate the detection of composition flaws. For the *gds* example we expressed the security policy in a propositional temporal logic, and modeled the components as a state-transition system. We then used a model checker to check first whether each component satisfies its local security policy and second, whether the composition of the components satisfies the given global security policy. A model checker either decides that the components individually and compositionally obey the respective security policies or give a counterexample execution that would show which sequence of actions could be taken to invalidate one of the policies. In the *gds* example the output of the model checker (SMV) was precisely the series of steps described in the attack section. Our hope is to allow automatic verification of the composition of software modules in order to detect flaws before the composed system is deployed. Moreover, by using model checking as a state exploration tool, we may be able to discover new attacks.

6. REFERENCES

[1] S. Nielson, S. J. Fogarty and D. S. Wallach. “Attacks on Local Searching Tools”. Technical Report TR04-445, Department of Computer Science, Rice University, December 2004.