Secure Deployment of Components

Mark Grechanik[1] and Dewayne E. Perry[2]

[1] Department of Computer Sciences,
University of Texas at Austin
gmark@cs.utexas.edu

[2] Department of Electrical and Computer Engineering,
University of Texas at Austin
perry@ece.utexas.edu

Abstract. The secure deployment of components is widely recognized as a crucial problem in component-based software engineering. While major effort is concentrated on preventing malicious components from penetrating secure systems, other security violations may also cause significant problems. We uncover a technique that creates a major breach of security by allowing rogue components to interfere with component-based applications by impersonating various generic components. This interference leads to stealing business value of competitive products and causes problems without violating legal agreements. We also present our solution to this problem, called *Secure COmponent Deployment Protocol (S-CODEP)*, and prove its soundness using the authentication logic of Burrows, Abadi, and Needham (*BAN authentication logic*).

1 Introduction

The secure deployment of components is widely recognized as a crucial problem in component-based software engineering (CBSE), and it has a major impact on the overall quality of component-based applications [1,2]. Component-based applications are ubiquitous in today's computing world. Many software vendors provide various generic components for free thereby reducing the cost and time required for development of commercial products that use these components. For example, Microsoft Windows comes with hundreds of generic components ranging from different GUI elements and FTP clients to the sophisticated Internet Explorer browser control.

The deployment of generic third-party components opens their users to different security risks. While major efforts are concentrated on preventing malicious components from penetrating secure systems, other security violations may also cause significant problems. Apart from general widely known security breaches like *buffer overflow* and *denial of service attack*, component deployment introduces CBSE-specific ways to exploit mechanisms incorporated in components and underlying component infrastructures. One of such security problem is the *impersonation* of some component by a surrogate component (e.g. inserting component server objects that act on behalf of their clients) [3] that serves as an *interceptor* [4,5,6] of all pa-

rameter values that clients of the impersonated component provide when invoking its interfaces and all values that these interfaces return to the clients after the execution.

We uncover a technique by which a rogue application legally installed by a user may impersonate a generic component by subverting existing security mechanisms in order to interfere with commercial component-based applications. The goal of this interference is to enable this rogue application to participate in the workflow of commercial applications thereby stealing the business value of competitive products and causing serious problems without violating legal agreements.

Suppose that application A sold by some company is a popular commercial product with a closed architecture that uses proprietary algorithms and techniques to deliver the business value to its users. Application A uses a generic component C that processes some proprietary data. On the other hand, application B is some inferior commercial product of a different company that would like to team up with the company that created product A. However, this partnership either costs company B some amount of money up-front that is B is unwilling to pay, or company A does not want such an alliance perceiving company B as A's potential competitor in the marketplace.

At this point, company B can use a security breach to integrate its application with A's product without violating legal agreements and without making any business alliance with company A. Company B creates a surrogate C' of component C with identical interfaces since all interfaces of C are well-documented. The public interfaces of C' only invoke real interfaces of C. That is, C' impersonates the component C and serves as an interceptor of all parameter values that A provides when invoking interfaces of C and all values that C returns to A after executing its interface methods. In fact, B can control the behavior of application A redirecting some functionality to itself thereby increasing its business value at the expense of A. We give a real-world example of this impersonation technique in Section 2.

When the user installs application B it deploys its surrogate component C' that replaces component C used by A. Since the user does not care about competitive differences between companies A and B s/he does not consider the above-mentioned action of B as a security breach providing that C' does not cause any harm to the computer (and no harm is caused except for a very small and mostly unnoticeable performance penalty due to the interception). Thus, company A may lose its competitive edge due to the security breach, and it cannot even sue company B since its actions do not violate standard license agreements that stipulate that only static modifications of executable programs produced by a company that sells a product, are prohibited.

Existing component and underlying distributed object infrastructures attempt to solve the problem of impersonation by providing special security settings that prohibit the further delegation of client requests by surrogate servers. This security approach can be easily subverted using various techniques that we review in Related Work.

Our solution is a protocol called *Secure COmponent Deployment Protocol (S-CODEP).* S-CODEP is based on the Kerberos [7] and a generic model of a component infrastructure that we present shortly. By identifying weak links of a component infrastructure in the deployment chain, we apply S-CODEP to eliminate these links. Finally, we prove the soundness of S-CODEP using the BAN authentication logic [8].

The contribution of this paper is thus in discovering a security breach when deploying components under certain settings, analyzing the sources of potential attacks that lead to this breach, proposing a protocol that eliminates this problem and formalizing it, and proving that our solution is sound.

2 A Real-World Motivating Example of the Impersonation

One of the authors (Grechanik) was a consultant to Globeset, Inc., an Austin, Texas based company that developed an infrastructure for secure electronic transactions. As part of this infrastructure the company developed software package called SSL Wallet that was supposed to assist customers with the automatic completion of electronic financial transactions. Globeset established business alliances with some large banks and merchants. These banks wanted customers that carried their credit cards to receive assistance from Globeset. On the other hand, merchants also believed that Globeset would help them to increase their sales since customers liked the automatic payment system, and the company would direct them to participating merchants for products they need.

Globeset ran into a problem when attempting to integrate its SSL Wallet with the AOL browser. AOL has its own proprietary Internet browser that in 1999 used Microsoft Internet Explorer control supplied with Windows as a generic component, around which most of the AOL browser functionality was built. With a customer base of over 20 million users, AOL had its own plans of developing an electronic payment wallet and signing agreements with major banks and merchants. Therefore Globeset received a definite “No” when it asked AOL to form a business alliance that would enable SSL Wallet to work seamlessly with the AOL browser.

Since modifications of the binary code of the AOL browser were prohibited under its license agreement, Grechanik exploited the fact that Globeset’s installation program and installed products were granted full administrative privileges by users who purchased SSL Wallet. He (with assistance of other developers) designed a surrogate control that impersonated Microsoft Internet Explorer component and provided information about the usage of the AOL browser at the runtime to the Globeset’s SSL Wallet. Once it became clear that a user wanted to buy a product, the SSL Wallet directed the user to a specific merchant’s web site and used credit card information to complete a transaction thereby effectively diverting business from AOL. The SSL Wallet was commercialized and successfully used by Globeset’s customers.

As of today this security breach is still intact, and to our knowledge no one identified it and took steps to eliminate this problem. We undertake this task with our solution.

3 Our Solution

Our solution consists of three parts. First, we present a model of a generic component infrastructure and analyze it to determine the weakest links that enable security

breaches such as our case of impersonation. Then, we describe the protocol S-CODEP that enables the secure deployment of components. Finally, we prove the soundness of S-CODEP using the BAN authentication logic.

3.1 Assumptions

We make four assumptions. First, we assume that the overall integrity of the operating system cannot be violated (i.e., no external threats can exploit general security breaches that may compromise the overall integrity of the system). It means that no program can modify the kernel of the operating system or change it by installing different system drivers that would otherwise compromise standard security settings. Second, we assume that when installing any program the operating system creates a sandbox effectively protecting the installation from being penetrated by an adversary. The third assumption is that digital certificates and private keys provided by certification authorities (for example, makers of components or third-party companies such as Verisign, Inc.) and carried by programs and components should be trusted and cannot be forged. Finally, no program installed by the user can analyze other programs to determine their control and data flows or can extract their computational logic (i.e. to reverse engineer installed software). Doing such analysis would require significant computational resources and is therefore impractical. It is noteworthy that these assumptions do not go beyond what is normally assumed about various security infrastructures.

3.2 Designators and Definitions

We use the following names and definitions throughout this paper. All principals are designated with uppercase English letters. We designate a commercial application as A and a rogue application that exploits A, as B. A generic component used by A is designated C, and its surrogate installed by B is designated as C'. A component infrastructure is designated by I, and an authentication server that is a part of the operating system is denoted by S. By a component we mean a logical unit of related data (e.g. a file) which comprises logically coupled compiled programs and the format of the file is recognized by the existing component infrastructure. The infrastructure uses a systemwide database with which the file is registered under a unique name, and independently deployed. A component model defines component interaction and composition standards.

4 Weak Security Links in Component Infrastructures

A component infrastructure is a set of system services based on a component standard that ensure the properties of components are immutable and enables rules of component composition and interaction. Known component infrastructures include DCOM/.Net, CCM, and Enterprise JavaBeans. In this paragraph we introduce a ge-

neric component infrastructure and analyze it on the subject of weak security links that enable a variety of sophisticated impersonation techniques for which existing security solutions are not adequate.

4.1 Generic Component Infrastructure

A generic component infrastructure model is shown in Fig. 1. It consists of the Component Loader, Classloader, a Finder Service, and a Systemwide Database. Since a component is often a file that comprises logically coupled compiled programs, the Component Loader reads the component file into memory, uncompresses it, and extracts individual programs. However, components are often located in a Systemwide Database that in a simplest case can be a file system. A client refers to a component by its unique name with which the component registers with the Systemwide Database. When the Component Loader receives a request from a Client to instantiate a component, it asks the Finder Service to locate the component.

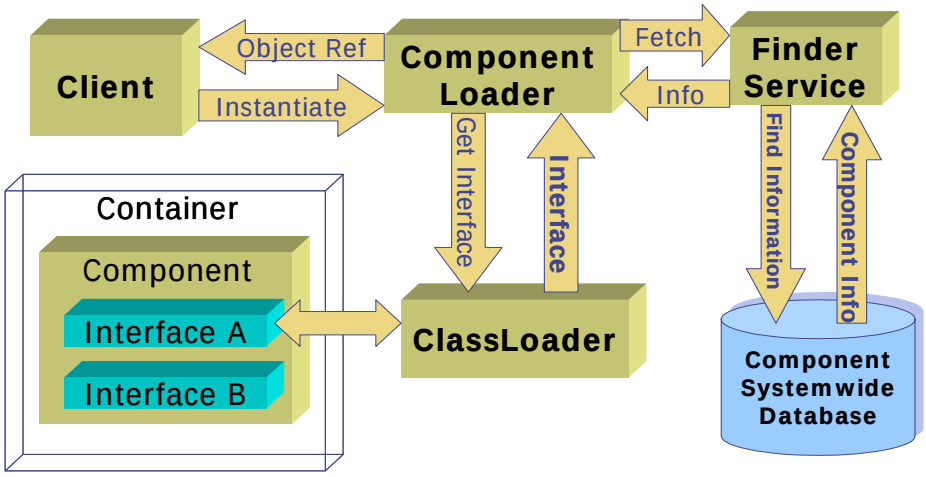

Fig. 1. A model of a generic component infrastructure and interactions among its elements.

When the Finder Service locates the component and provides information about its location, the Component Loader loads this component and asks the Classloader to instantiate classes that provide interfaces needed by the Client. The Classloader accomplishes this job and sometimes instantiates a special library called a Container that provides generic interfaces for certain classes of components. When all classes are instantiated the Component Loader returns a reference to the component that the Client can use to call exposed interfaces for the duration of the session.

4.2 Architectural Security Analysis

Here we perform an architectural security analysis of the generic component infrastructure presented in the previous section. We use the strategy described in [3] for identifying security risks. As the first step we gather information sufficient to understand the structure of the system at a high level (which we did in the previous section), and then we consider how this system operates in a concrete environment.

Concrete implementations (e.g., DCOM, CCM, Enterprise JavaBeans) consist of different modules implementing basic elements of a component infrastructure. These modules are not a part of the underlying operating system and often are user-level programs similar to A and B. It means that B can use binary rewriting [9,10,11] to dynamically modify infrastructure modules or impersonate some of their services similar to the technique for impersonating components. Essentially, if we build an attack tree (i.e., a graph representing the decision-making process of well-informed attackers), then we obtain a large number of possible paths that lead to achieving the security breach by impersonating each element of the generic component infrastructure and then intercepting calls to its interfaces.

5 S-CODEP

S-CODEP is our solution to the problem of the component impersonation security breach created by rogue programs. We base S-CODEP on the same idea that lies in the foundation of Kerberos protocol [7] – to establish shared keys between principals with help from an authentication server. Since the operating system cannot be compromised (by our first assumption), we can use its security layer to act as an authentication server. By our second assumption a program cannot be compromised during its installation. These and other assumptions allow us to establish a secure communication channel between the principal A and the component C that ensures the true identities of the participants of this protocol.

5.1 Description of S-CODEP

S-CODEP exploits the second assumption of secure installation to obtain shared keys from component infrastructure I. During the installation, each principal P sends a message to server S (we mean a security service within an operating system) that contains its identity and a private key that P uses to communicate with S: $\{P, K_p\}$. This operation includes the installation of components. Recall that by our third assumption no program can attempt to reverse engineer other programs to obtain private keys that are used to communicate with S. When S receives this initial message it stores it in a secure storage to which only S can gain access since by our first assumption S cannot be compromised.

A first step is to establish a secure communication channel between A and I so that I can request various services from components securely. To do that, A sends a message encrypted with K_A to S in which it asks S to provide shared key K_{AI} to commu-

nicate with component infrastructure I. S receives this message from A and decrypts it with K_A^{-1}. It responds to A with a message encrypted with K_A that contains the session key Ts, shared key K_{AI}, and a message M_{IS} encrypted with K_I that contains the session key Ts, shared key K_{AI}, and the identity of A.

A receives this message from S and decrypts it with K_A^{-1}. It initiates a communication process with I by sending a message encrypted with the shared key K_{AI} to I. This message contains the message M_{IS} encrypted with K_I that A received from S, and a message encrypted with the shared key K_{AI} that contains the identity of A and a timestamp nonce T_A. I receives this message from A, decrypts it, and responds with a message that consists of the incremented timestamp nonce T_A encrypted with the shared key K_{AI}. With reception of the last message from I a secure communication channel is established between A and I.

Suppose that program A needs to instantiate C at some point of time. Using the established secure communication channel with I, A sends a message to I encrypted with the shared key K_{AI} that contains the incremented timestamp nonce T_A, unique identifier of the requested component C, and the description of the requested operation (e.g., to instantiate this component). After receiving and decrypting this message I asks permission from S to establish a secure communication channel with C. That is, I sends a message to S encrypted with K_I that contains I's and C's identities as well as the session key Ts. In this message I asks S to provide a shared key K_{IC} to communicate with component C. S receives this message from I and decrypts it with the key K_I^{-1}. It responds to I with a message encrypted with K_I that contains shared key K_{IC}, and a message M_{CS} encrypted with K_C that contains the session key Ts, the shared key K_{IC}, and the identity of I.

I receives this message from S and decrypts it with K_I^{-1}. It initiates a process of instantiating C by establishing a secure communication channel with C. It does so by sending a message encrypted with the shared key K_{IC} to C. This message contains the message M_{CS} encrypted with K_C that I received from S, and a message encrypted with the shared key K_{IC} that contains the identity of I and a timestamp nonce T_I. C receives this message from I, decrypts it, and responds with a message that consists of the incremented timestamp nonce T_I and reference R_C to the component C, encrypted with the shared key K_{IC}. With reception of the last message from C a secure communication channel is established between I and C. I extracts the reference R_C to the component C and forms a new message comprising the incremented timestamp nonce T_A, unique identifier of the requested component C, and the reference R_C and encrypts this message with the shared key K_{AI}. Then it sends this message to A. After decrypting it A holds the reference R_C to the component C.

When A needs to invoke some interface of C, it sends a message to I that is encrypted with the shared key K_{AI} that contains the incremented timestamp nonce T_A, the reference R_C to the component C, the list of parameter values in the order in which they are specified in the declaration of this interface, and the description of the requested operation (e.g., to invoke an interface of the component). I decrypts this message and invokes the requested interface on behalf of A, and receives the return values. Then it forms a message encrypted with the shared key K_{AI} that contains the incremented timestamp nonce T_A, the reference R_C to the component C, the list of

return values, and the description of the requested operation. This concludes the description of S-CODEP.

5.2 BAN Logic Notation

In order to prove that S-CODEP is sound we need to formalize it. We selected BAN formalism [8] for this purpose that is built on a many-sorted model logic. This logic comprises following sorts of objects: principals, encryption keys, and formulas. Principals are designated with uppercase English letters, and K_{AB} denotes a shared key between principals A and B. The symbols P and Q range over principals; X and Y range over statements; and K ranges over encryption keys.

BAN defines following constructs:

- $\mathbf{P}$ **believes** $\mathbf{X}$: the principal $\mathbf{P}$ may act as if $\mathbf{X}$ is true;
- $\mathbf{P}$ **sees** $\mathbf{X}$: $\mathbf{P}$ receives and reads message $\mathbf{X}$ after decrypting it;
- $\mathbf{P}$ **said** $\mathbf{X}$: $\mathbf{P}$ sent a message containing statement $\mathbf{X}$ at some time;
- $\mathbf{P}$ **controls** $\mathbf{X}$: $\mathbf{P}$ has jurisdiction over $\mathbf{X}$ meaning that the principal $\mathbf{P}$ is an authority on $\mathbf{X}$ and should be trusted in this matter;
- **fresh**($\mathbf{X}$): $\mathbf{X}$ has not been sent in a message at any time before the current run of the protocol;
- $\mathbf{P} \overset{K}{\longleftrightarrow} \mathrm{Q}$: $\mathbf{P}$ and $\mathbf{Q}$ use shared key $\mathbf{K}$ to communicate and this key can never be discovered by any principal except for $\mathbf{P}$ and $\mathbf{Q}$;
- $\mathbf{P} \overset{X}{\rightleftharpoons} \mathrm{Q}$: Only $\mathbf{P}$ and $\mathbf{Q}$ and principals they trust know secret $\mathbf{X}$;
- $\{X\}_K$: an abbreviation for an expression that $\mathbf{X}$ is encrypted using key K, and
- $\langle X \rangle_Y$: $\mathbf{Y}$ is a digital certificate or password that is combined with $\mathbf{X}$ to prove the identity of the sender of this message.

Authentication protocols are described as sequences of messages, each message is written in the form $\mathbf{P} \longrightarrow \mathrm{Q}$: message meaning that principal $\mathbf{P}$ sends message to the principal $\mathbf{Q}$.

5.3 BAN Logical Postulates

The authentication logic of BAN is based on a set of postulates that we use to prove certain properties of S-CODEP. Here we describe these postulates and explain their meanings.

A shared key postulate expressed in (Eq 1) states that if P believes that the key K is shared with Q and sees X encrypted under K, then P believes that Q once said X. This rule is also called a *message meaning rule*.

$$\frac{P \text{ believes } Q \xleftrightarrow{K} P \quad P \text{ sees } \{X\}_K}{\vdash P \text{ believes } Q \text{ said } X} \qquad \textbf{(Eq 1)}$$

A variant of this rule for shared secrets is expressed in (Eq 2) and states that if P believes that the secret Y is shared with Q and sees $\langle X \rangle_Y$, then P believes that Q once said X.

$$\frac{P \text{ believes } Q \overset{Y}{\rightleftharpoons} P \quad P \text{ sees } \langle X \rangle_Y}{\vdash P \text{ believes } Q \text{ said } X} \qquad \textbf{(Eq 2)}$$

Nonce is a term used to describe a message whose purpose is to be fresh, i.e., this message was not sent at any time before the current run of the protocol. A postulate expressed in (Eq 3) is called *the nonce-verification rule* and states that if P believes that X has been sent in the present run of the protocol and that Q once said X, then P believes that Q believes X.

$$\frac{P \text{ believes fresh}(X) \quad P \text{ believes } Q \text{ said } X}{\vdash P \text{ believes } Q \text{ believes } X} \qquad \textbf{(Eq 3)}$$

(Eq 4) is the jurisdiction rule stating that if P believes that Q has jurisdiction over X then P trusts Q on whether X is true.

$$\frac{P \text{ believes } Q \text{ controls } X \quad P \text{ believes } Q \text{ believes } X}{\vdash P \text{ believes } X} \qquad \textbf{(Eq 4)}$$

(Eq 5) expresses the fact that if P sees an encrypted message, then it can also see its content. Finally, (Eq 6) states that if any part of a message is fresh, then the entire message is also fresh.

$$\frac{P \text{ believes } Q \xleftrightarrow{K} P \quad P \text{ sees } \{X\}_K}{\vdash P \text{ sees } X} \qquad \textbf{(Eq 5)}$$

$$\frac{P \text{ believes fresh}(X)}{\vdash P \text{ believes fresh}(X,Y)} \qquad \textbf{(Eq 6)}$$

5.4 Formalization of S-CODEP

At this point we are ready to provide a formal description of S-CODEP. The protocol consists of ten mandatory initial messages whose purpose is to establish secure communication channel between A and I. Once this task is accomplished, A proceeds to request invocations of C's interfaces and I returns the results of these invocations. Idealized messages that accomplish this task of interface invocations are shown below as Messages N and N+1.

Message 1: $\mathbf{A} \longrightarrow \text{S: } \{A, I\}_{K_A}$

Message 2: $\mathbf{S} \longrightarrow \text{A: } \left\{T_s, K_{AI}, \{T_s, K_{AI}, A\}_{K_I}\right\}_{K_A}$

Message 3: $\mathbf{A} \longrightarrow \text{I: } \{T_S, K_{AI}, A\}_{K_I}, \{A, T_A\}_{K_{AI}}$

Message 4: $\mathbf{I} \longrightarrow \text{A: } \{T_A + 1\}_{K_{AI}}$

Message 5: $\mathbf{A} \longrightarrow \text{I: } \{T_A + 2, C, \text{instantiate}\}_{K_{AI}}$

Message 6: $\mathbf{I} \longrightarrow \text{S: } \{I, C, T_S\}_{K_I}$

Message 7: $\mathbf{S} \longrightarrow \text{I: } \left\{K_{IC}, \{T_S, K_{IC}, I\}_{K_C}\right\}_{K_I}$

Message 8: $\mathbf{I} \longrightarrow \text{C: } \{T_S, K_{IC}, I\}_{K_I}, \{I, T_I\}_{K_{IC}}$

Message 9: $\mathbf{C} \longrightarrow \text{I: } \{T_I + 1, R_C\}_{K_{IC}}$

Message 10: $\mathbf{I} \longrightarrow \text{A: } \{T_A + 3, C, R_C\}_{K_{AI}}$

……

Message N: $\mathbf{A} \longrightarrow \text{I: } \left\{T_A + N + 3, R_C, \{p_1, p_2, \ldots, p_q\}, \text{invoke(interface)}\right\}_{K_{AI}}$

Message N+1: $\mathbf{I} \longrightarrow \text{A: } \left\{T_A + N + 4, R_C, \{r_1, r_2, \ldots, r_t\}, \text{invoke(interface)}\right\}_{K_{AI}}$

Message N+2: $\mathbf{A} \longrightarrow \text{I: } \{T_A + N + 5, R_C, \text{terminate}\}_{K_{AI}}$

Formalization of S-CODEP allows us to make assumptions that we use to prove the soundness of this protocol. Messages 1 to 4 establish the fact (A 1) that principal P communicating with component infrastructure I believes that I and P communicate using shared key K_{PI}. Protocol messages 5 to 10 ensure that the principal P receives messages from the real component C via I and reads them after decrypting using key K_{PI}. We specify this fact in our assumption (A 2). Since we include encrypted none

T_C in each message, we state that the freshness of messages that P sees is always guaranteed (A 2). Finally, since I is responsible for handling the lifecycle of C we state that P believes that I has the authority over C in our assumption (A 2).

$$\text{P believes I} \xleftrightarrow{K_{PI}} \text{P} \qquad \textbf{(A 1)}$$

$$\text{P sees } \{T_C, C\}_{K_{PI}} \qquad \textbf{(A 2)}$$

$$\text{P believes fresh}(T_C) \qquad \textbf{(A 3)}$$

$$\text{P believes I controls C} \qquad \textbf{(A 4)}$$

5.5 Proof of Soundness of S-CODEP

Given S-CODEP we need to be sure that every time a principal invokes interfaces of component C it actually communicates with the real component C and not some intercepting surrogate C'. Proving this property is in fact equal to establishing that the model of the protocol is sound, that is, it does not lead to wrong behavior when some intercepting surrogate C' can impersonate the real component C.

Theorem. When principal P communicates with C it believes that C is true (i.e., P believes C).

Proof.

We start with the assumptions (A 1), (A 2), (A 3), and(A 4). Here T_C stands for nonces used in S-CODEP as part of secure communications between A, I, and C. By applying the BAN message meaning rule (Eq 1) and the assumptions (A 1) and (A 2), we derive

$$\frac{\text{P believes I} \xleftrightarrow{K_{PI}} \text{P} \quad P \text{ sees } \{T_C, C\}_{K_{PI}}}{\text{|-- P believes I said } \{T_C, C\}} \qquad \textbf{(Eq 7)}$$

In the next step of our proof we take assumption (A 3) and apply (Eq 6) to obtain the following result shown in (Eq 8).

$$\frac{P \text{ believes fresh}(T_C)}{\text{|-- } P \text{ believes fresh}(T_C, C)} \qquad \textbf{(Eq 8)}$$

In the penultimate step of our proof we take rules (Eq 3), (Eq 7), and (Eq 8) and obtain

$$\frac{P \text{ believes fresh}(T_C,C) \quad P \text{ believes I said } \{T_C,C\}}{\vdash\!\!- \; P \text{ believes I believes } \{T_C,C\}} \qquad \textbf{(Eq 9)}$$

Finally, we take assumption (A 4) and inference rule (Eq 4) and obtain

$$\frac{P \text{ believes I controls } \{T_C,C\} \quad P \text{ believes I believes } \{T_C,C\}}{\vdash\!\!- \; P \text{ believes } \{T_C,C\}} \qquad \textbf{(Eq 10)}$$

which is the statement of the theorem we need to prove. □

6 Related Work

While the areas of component deployment and software security are, each on its own, rife with excellent publications, the intersection of these areas of research yields few but memorable results. It is widely recognized that no single technique can produce completely trusted components [12]. The uniqueness of this paper is in identifying a problem that leads to creation of untrustworthy components and in using sophisticated algorithms developed in the area of secure communication protocol to solve it.

The Trusted Components Initiative (TCI) is a cooperative effort to provide the software industry with methods, techniques and tools for building high-quality reusable components, thereby elevating the general level of trust that software users, and society at large, can have in these components [13]. TCI's web site contains a page with references to the growing number of publications in this area.

One of main research directions in secure component deployment is in providing strong authentication control of access to and from components [14,15]. A number of techniques and algorithms are designed to prevent malicious components to gain access to computers inside secured networks, or to prevent components to access domains for which they do not have a proper authorization. However, these solutions fall short to solve the problem that we pose in this paper since this trick exploits the inability of component infrastructures to prevent two programs with equally high security privileges from interfering with each other via a commonly used generic component.

Software fortresses is an approach for modeling large enterprise systems as a series of self-contained entities [16]. Each such entity is called a fortress and it makes its own decisions about the underlying platform, data storage, and security mechanisms and policies. Fortresses communicate with one another through carefully designed mechanisms. Once a component is allowed inside a fortress it gains access to all other components within the same fortress. No mechanisms are offered to solve the impersonation problem that we presented in this paper.

Existing component and underlying distributed object infrastructures attempt to solve the problem of impersonation by providing special security settings that prohibit the further delegation of client requests by surrogate servers. For example, DCOM [17] defines four impersonation levels:

- Anonymous level completely hides the credentials of the caller;

- Identify level enables objects to query the credentials of the caller. Under this level access control lists are checked and no scripts can be run against remote computers;
- Impersonate level enables objects to use the credentials of the caller, and
- Delegate level enables objects to permit other objects to use the credentials of the caller.

This security approach can be easily subverted using various techniques. It is noteworthy that impersonation is a useful technique and is often required to implement different software architectures [6] so turning it off may not be allowed at all. However, even when security settings are explicitly set to prohibit impersonation it may still be possible to implement it with clever tricks. For example, since programs that are installed on the same computer under high administrative privileges may modify security settings during the installation, the security attribute prohibiting impersonation can be simply turned off. A more complicated example of bypassing security setting that prohibits impersonation is installing a daemon that instantiates a real component and communicates with a surrogate component by using some interprocess communication channel.

CORBA and CCM implementations may use security services defined by the Object Management Group's standards [18]. Authorization Token Layer Acquisition Service (ATLAS) describes the service needed to acquire authorization tokens to access a target system using the CSIv2 protocol. This design defines a single interface with which a client acquires an authorization token. This token may be pushed, using the CSIv2 protocol in order to gain access to a CORBA invocation on the target. This specification solves the problem of acquiring the privileges needed for a client to acquire a set of privileges the target will understand.

The Common Secure Interoperability Specification, Version 2 (CSIv2) defines the Security Attribute Service that enables interoperable authentication, delegation, and privileges. CORBA Security Service provides a security architecture that can support a variety of security policies to meet different needs. The security functionality defined by this specification comprises a variety of mechanisms such as identification and authentication of principals and authorization and infrastructure based access control. While CORBA and CCM security specifications provide a protection against the impersonation threat in general by introducing different authentication levels similar to DCOM, this protection can be easily subverted using the techniques described above.

Some component infrastructures make the task of impersonation of components simpler. A common technique in Windows platforms is emulation [19]. Each COM component (or dynamically linked library (DLL)) has a GUID (Globally Unique IDentifier). The Windows registry is a database of DLLs organized by GUIDs. A client loads a DLL by invoking a system call with the DLL's GUID; the registry is searched for the location of that DLL. Emulation is a registry technique that replaces an existing DLL with an intermediary DLL that implements the same COM interfaces and has the same GUID. When a client requests a particular DLL, the intermediary is loaded instead. In turn, the intermediary loads the shadowed DLL and acts as a "pass-through" for all client requests. Thus the intermediary can monitor call traffic for a COM interface unobtrusively.

7 Conclusion

Secure deployment of components is widely recognized as one of the crucial problems in component-based software engineering. The contribution of this paper is first in discovering a technique that creates a major breach of security by allowing rogue components to interfere with component-based applications by impersonating various generic components. This interference leads to stealing business value of competitive products and causes serious problems without violating legal agreements. Our second contribution is a solution to this problem that we call Secure COmponent Deployment Protocol (S-CODEP). We prove its soundness using the authentication logic of Burrows, Abadi, and Needham (BAN authentication logic). We know of no other work that addresses the problem that we present in our paper and solves it.

Acknowledgments. We warmly thank Don Batory for reading this paper and providing useful comments, suggestions, and corrections.

References

1. Meyer, B.: The Grand Challenge of Trusted Components. The 25th International Conference on Software Engineering, Portland, OR, (2003)
2. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. ACM Press, Addison-Wesley (1998)
3. Viega, J. and McGraw, G.: Building Secure Software. Addison-Wesley (2002)
4. Brown, K.: Building a Lightweight COM Interception Framework, Part I: The Universal Delegator. Microsoft Systems Journal, 14 (1999) 17-29
5. Brown, K.: Building a Lightweight COM Interception Framework, Part II: The Universal Delegator. Microsoft Systems Journal, 14 (1999) 49-59
6. Schmidt, D., Stal, M., Rohnert, H., and Buschman, F.: Pattern-Oriented Software Architecture. John Wiley & Sons, 2 (2001) 109-140
7. Tung, B.: Kerberos: A Network Authentication System. Addison-Wesley (1999)
8. Burrows, M., Abadi, M., and Needham, R.: A Logic of Authentication. ACM SIGOPS Operating Systems Review 23(5) 1989
9. Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., and Bershad, B.: Instrumentation and Optimization of Win32/Intel Executables Using Etch. USENIX Windows NT Workshop, Seattle, WA (1997)
10. Hunt, G.: Detours: Binary Interception of Win32 Functions. Proc. 3rd USENIX Windows NT Symposium, Seattle, WA (1999)
11. Larus, J. and Schnarr, E.: EEL: Machine-Independent Executable Editing. SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (1995)
12. Meyer, B., Mingins, C., and Schmidt, H.: Providing Trusted Components to the Industry. IEEE Computer (1998) 104-115
13. The Trusted Components Initiative: http://www.trusted-components.org/
14. Bagarathan, N. and Byrne, S.: Resource Access Control for an Internet User Agent. The 3rd USENIX Conference on Object-Oriented Technologies and Systems (1997)
15. Lindqvist, U., Olovsson, T., and Jonsson, E.: An Analysis of a Secure System Based on Trusted Components. Proceedings of 11th Ann. Conf. Computer Assurance, (1996) 213-223
16. Sessions, R.: Software fortresses : modeling enterprise architectures. Addison-Wesley, (2003)
17. Brown, N. and Kindel, C.: Distributed Component Object Model Protocol - DCOM/1.0. Internet Draft, January 1996. http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt
18. Object Management Groups security standards: http://www.omg.org/technology/documents/formal/omg_security.htm
19. MSDN Library: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnesscom/html/classemulation.asp