

# COTS, the safety nightmare of component-oriented frameworks

*Lieven Desmet, Liesbeth Jaco,  
Koenraad Mertens, Tine Verhanneman*

*Report CW 367, September 2003*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# COTS, the safety nightmare of component-oriented frameworks

*Lieven Desmet, Liesbeth Jaco,  
Koenraad Mertens, Tine Verhanneman*

*Report CW 367, September 2003*

Department of Computer Science, K.U.Leuven

## **Abstract**

Third party components seem to be an easy solution when implementing a sophisticated software application. This paper takes a closer look at the security implications for a system that uses third party components. We will focus on third party components for which the source code is not available and thus the exact behavior is unknown. Different methods that can be used for testing and validating the given specifications as well as the undocumented behavior of these components will be discussed. On-line as well as off-line methods will be presented. We will conclude that no single method provides an acceptable degree of security and, because the testing and validation effort must remain lower than designing and producing the component for yourself, not even the combination of all possible methods provides a sufficient degree of security for a mission critical application.

# COTS, the safety nightmare of component-oriented frameworks

Lieven Desmet, Liesbeth Jaco, Koenraad Mertens, Tine Verhanneman  
DistriNet, Dept. of Computer Science, K.U.Leuven

DistriNet Workshop on Run-Time Adaptation in Distributed Software  
November 29-30, 2002

## **ABSTRACT**

*Third party components seem to be an easy solution when implementing a sophisticated software application. This paper takes a closer look at the security implications for a system that uses third party components. We will focus on third party components for which the source code is not available and thus the exact behavior is unknown. Different methods that can be used for testing and validating the given specifications as well as the undocumented behavior of these components will be discussed. On-line as well as off-line methods will be presented. We will conclude that no single method provides an acceptable degree of security and, because the testing and validation effort must remain lower than designing and producing the component for yourself, not even the combination of all possible methods provides a sufficient degree of security for a mission critical application.*

## **1 Introduction**

The use of components is becoming increasingly important and wide-spread: they enable a cost-effective composition of the system, since every vendor only needs to concentrate on the development of a particular aspect of the system. Moreover, components can be reused as they are developed as independent units of the framework. This paper focusses especially on the use of third party component from which the source is unavailable, *COTS*<sup>1</sup> components.

New security issues arise when deploying such third party components, because new principals and roles are involved. In addition to users and application system owners, one also needs to take into account the component vendors and host providers. Appropriate protection is needed for the applications and their systems on the one hand and for the component vendor on the other hand:

Applications and systems need to be protected against malicious components: memory safety,

stack safety, flow safety, ... The threat that a flow of data or events may be changed, so that data or events are forwarded to a component without the required access rights, is particularly relevant in distributed component-structured systems, since those systems have different user-access policies on the various computers on the network.

The component vendor needs to be protected against wrong incriminations, malicious surrounding components and malfunctions of hosting environments, unlicensed employment of components, stealing of data initialisation and source code.

Another recent evolution in component-oriented software, is the need for run-time adaptation and customization. Framework support enables the hot-swapping of components within the application. During the deployment of the application, components could easily be plugged out and plugged in, without the need for bringing the application off-line.

In this paper we will mainly focus on the security of the application, when third party components are integrated at run-time. We will give a short overview of possible security solutions for the construction of a safe and reliable system and we will evaluate them in the context of a run-time adaptive environment.

We will conclude this evaluation by stating that none of the presented techniques will entirely eliminate the security problems in a reasonable fashion, even not when used in combination with each other. This is an important consideration when deciding to use third party components in mission critical systems, because the entire framework of components is just as safe as the least safest component in the entire network.

## **2 Possible security techniques**

By and large, the techniques can be divided into three categories, which will be presented in this section: code instrumentation, trust-based relationships and component testing. The first two techniques will be shortly enumerated, so that we can

---

<sup>1</sup>Commercial Off The Shelf Software

elaborate on the last one.

## 2.1 Code instrumentation

The first category contains techniques that alter the machine code in a way that critical operations can be analyzed before or monitored during the execution of the code in order to detect attacks and security deficiencies. These techniques are enforced by the framework or the programming language. The next paragraph describes briefly three categories of code instrumentation: fault isolation, security automata and language based security.

Firstly, the framework can provide *fault isolation*, which allows non-trusted code to execute in a safe system part where it cannot cause damage and it can be monitored. An example is the java sandboxing approach. This technique seems very safe, although – in practice – it could be very difficult to separate a small part of an application into a different sandbox without creating a lot of overhead and without disrupting the needed interaction between different components.

An alternative approach enforces the security policy by means of *security automata*: the code is only allowed to execute if the next step corresponds to a transition in the automaton. This technique hardens a system quite effectively, but is hardly applicable in a real system, since the construction of a complete state diagram of an adaptive system with enough internal state information is quite complex.

As a last example, *language based security* provides security-related information, obtained during parsing or other programming analysis. This information can be used to check whether the code complies to its security policy. The Java security architecture, Proof Carrying Code and .NET library interception belong to this category of techniques. The current security policies are not instantiation specific and do not cover interaction between objects. Therefore, security policies are only part of the solution.

## 2.2 Trust-based relationships

The second category of security techniques for constructing a secure framework, consists in trusting that the component will behave as specified or as certified. This trust can be established directly or distributed.

*Specification trust* takes in various forms: one can trust the vendor, the software component itself, the contract of the component (specified preconditions, postconditions, required security permissions, possible state transitions, ...).

*Certification trust* is used to establish trust in a component by relying on a trusted third-party certifier or a distributed certifying network.

Although these techniques of trust relationships are widely used, we state that you cannot rely only on such trust. Trusted specifications or certifications handle trust of components in isolation. Whenever such components are used, extra integration problems could appear. Therefore the *Trust Information service* should be extended with a *Conflict Resolution Service* and extra integration testing. Also, certified component contracts are not enough elaborated on and standardized.

## 2.3 Component testing

Instead of using code instrumentation or trust relationships, the last category of security techniques uses bare testing to improve the security and reliability of a system in the adoption of third party components.

**Different test-stages** Tests could be done at different stages: isolation testing, integration testing and system testing.

As a first step, the component could be tested *in isolation*. This test verifies the quality of the specific component and checks the promised functional behavior.

In a second stage, the integration of the component into an application could be tested. Even if the specific component is of high quality, there is no guarantee that the application will tolerate the component. If the application environment does not adopt the integrated component, undesirable integration problems could arise. Therefore *integration testing* tests the integration of a specific component in combination with other components.

In a third stage, the functionality of a complete application, with integrated third party components, could be assessed. Thereby the complete system is verified to behave correctly.

**Software wrapping** The approach for testing is based on the notion of software wrapping.

Software wrapping allows the data, flowing in and out of the component at the public interface level, to be intercepted. Also communication with other components and events is examined before passing through. Software wrapping isolates software components from the rest of the application and vice versa. It is an additional software layer to encase the component to be tested and verified.

The actual tests take place in between the external interface of the wrapper and the internal interface of specific components. Input could be rejected or adapted and output could be verified or dropped. Once a component is isolated, the testing process

could try to understand whether or not the component is interacting with the rest of the system as expected.

Software wrappers are transparent for the application and the wrapped software components. They have the same interface as the wrapped software and could easily be plugged around or removed from software components.

**Off-line and online testing** In the next section, a distinction is made between off-line testing and online testing.

In *off-line testing*, testing is done without any disturbance of the running application. A replica of the running application is built, or complete isolated tests are performed.

In contrast, *online testing* is done by assessment of specific components in the running (production) application. Messages passing in the running application are directly influenced by the components, which are tested.

### 2.3.1 Off-line testing

Off-line tests don't interfere with a running application. The tests are performed in advance to guarantee its well-behavior. In the next paragraphs, three examples of off-line testing are discussed: black box testing, fault injection and replica system testing.

The component must be analyzed and tested as a *black box*, since the source code of a COTS component is often unavailable. Even more, sometimes the vendor fails to provide a correct or complete description of the COTS component's behavior.

Black box testing starts without any knowledge of how the component works internally. We only know how to address the component's interface, and have some knowledge of what output should be expected. Black box tests assert the description of the functional behavior of the COTS component. Those assertions could be either of components in isolation (isolation tests) or of components during integration (integration tests). If the description fails, either the description is adapted, or a software wrapper is implemented for extra input and output checks, since the core functionality of a COTS component cannot be adapted.

Black box testing is (next to white box testing) used on a regular basis in traditional software testing.

*Fault injection* consists in passing corrupted data between components. This type of testing presents the system scenarios not found in typical testing. Instead of addressing problems related to defective software, fault injection relates to defective software environments.

By using fault injection to analyze software components, it is possible to prepare for the worst possible outcomes and anticipate them in the production

system. Fault injection is a good test procedure for verifying the robustness of the off-line system and specifically the software components, without the risk of bringing your online production system in any danger.

In the previous paragraphs, components were tested in isolation and during integrating. In this paragraph *system replica tests* analyze the complete system behavior.

Since online testing could endanger running applications and since tests on complete systems could give better feedback on integration and combination of components, the following approach is suggested: instead of testing the integration of the component on a running system, a complete copy of a running system infrastructure is made. Thereby not only the complete structure of the connected components is crucial, but also framework support is needed to copy the internal states of every component in the application (a system state transfer solution).

With this detailed copy of structure and internal state, a complete *off-line* system replica could be built, ready for testing purposes. In a realistic scenario, a message stream could be recorded at the input and the output of the online system to be replayed in the off-line system replica.

This technique enables complete system testing in an environment that is very similar to the actual online system, without any risk for the mission critical online system. Of course, extra framework support is needed in order to make the complete copy (structure and internal states). Therefore, further investigation on this technique is needed.

### 2.3.2 Online testing

Online tests act on running applications. Therefore, those tests are limited because they may not interfere with the normal functioning of the system. In the next paragraphs, two online test procedures are described: employment agreement tests and data collection.

In the first case, the wrapper around the used COTS component tests the *employment agreement*. The employment agreement is the application and component specific agreement between the application and the component considering the agreed behavior of the component in that application. The agreement is a matching between the needs of the application and the specification of the component. Tests within the wrapper check if the component *and* the application cooperate as agreed. Two different wrapper approaches are used.

One approach blocks inputs to the component which do not conform to the *preconditions* of the component's specifications, so that the component is prohibited to execute on those inputs.

The other approach is to capture the output be-

fore the component releases it, to check that the *postconditions* are satisfied, and to allow only those outputs that satisfy those constraints.

Another online approach, is to use wrappers around the components, not for online testing purposes, but for data collection. While running, all input, output, and events are logged into a database. The collected data could be analysed off-line. The advantage of this technique is that there is no interference with the running applications. The major disadvantage is that there is no direct protection. The application owner can only react when the evil has already taken place.

### 2.3.3 Evaluation remarks

In this section we make some remarks about the use of component testing techniques for the validation of third party components. These remarks are perfect starting points for future discussions about secure integration of COTS components:

- The complexity of possible integration problems, and the comprehensiveness of off-line techniques like black box testing and fault injection, suggest that combination of off-line and online tests are a necessity to provide a certain level of behavior validation.
- Although combination of off-line and online system is suggested in the previous paragraph, adaptive systems will, in extremis, adopt components at run-time, just seconds before using them. Therefore, online testing techniques need to be further elaborated. And in such extreme adaptive systems, testing should be combined with other security techniques like code instrumentation and trust-based relationships.
- An important remark to keep in mind while using testing techniques for validation, is that the effort to test a third party component (program effort for constructing the testing wrapper, run-time overhead for the tests, ...) may not exceed the effort to efficiently code the component yourself. Therefore testing techniques should be combined with trust-based techniques and code instrumentation to diminish the needed degree of testing.
- Since writing a wrapper to test a specific component is nothing more than combining some generic tests into a specific wrapper, a generic testing framework should be used to simplify the testing procedure.

## 3 Conclusion

The increasing use of third party components in component-oriented frameworks introduces a new

set of security issues. In this paper we focused on possible techniques for run-time integration of such components, especially in mission critical systems, with respect for memory safety, stack safety, flow safety, ...

Code instrumentation techniques (fault isolation, security automata, language based security, ...) are theoretically very secure. In practice however, those techniques are still too immature to handle all security issues or imply a lot of integration and run-time overhead.

Trust-based relationships try to solve the problem of malicious components by trusting vendors or applications at the one hand and by using trusted certifiers at the other hand. Although trust relationships are widely used, this solution does not cover the problem of conflicts between components at integration time and of a mismatch between the certifier's test policy and your particular application requirements.

The problem of integration conflicts or component validation can be solved by testing the component yourself. You can test the software far away from your running application or on your application itself or even better, a combination of both techniques.

Testing every component yourself seems to be the ideal solution. However the problem is you have to check every possible input and output, even the most unexpected, always have to keep an eye on every possible side effect, and have to make sure your online system keeps on going during the complete testing process. Off-line testing can help, but who says no input you consider safe and valid for a certain COTS component could not trigger an unexpected event at any future system state in the application ?

The conclusion of this paper is that, with the current state-of-the-art, you cannot safely rely on any third party component, not without combining so many techniques for testing and validation, not without thinking of so many unlikeliness, not without creating so many run-time overhead, in any better way than writing the component yourself.

## References

- [1] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins, *Making components contract aware*, IEEE Software, June 1999, pp. 38–45.
- [2] Jennifer Haddock, Gregory Kapfhammer, and Christoph Michael, *An approach for understanding and testing third-party software com-*

- ponents*, 48th Reliability and Maintainability Symposium. Seattle, WA, January 2002.
- [3] Peter Herrmann and Heiko Krumm, *Trust-adapted enforcement of security policies in distributed component-structured applications*, Proceedings of the 6th IEEE Symposium on Computers and Communications, Hammamet, IEEE Computer Society Press, July 2001, pp. 2–8.
  - [4] Peter Herrmann, Lars Wiebusch, and Heiko Krumm, *State-based security policy enforcement in component-based e-commerce*, Proceedings of the 2nd IFIP Conference on E-Commerce, E-Business, and E-Government (I3E), October 2002.
  - [5] Ulf Lindqvist and Erland Jonsson, *A map of security risks associated with using COTS*, Computer **31** (1998), no. 6, 60–66.
  - [6] Hardy Matthew, *Cots components in software development*, Proceedings of the Computer Science Discipline Seminar Conference (CSCI 3901).
  - [7] Sam Michiels, Dirk Walravens, Nico Janssens, and Pierre Verbaeten, *DiPS: Filling the Gap between System Software and Testing*, Workshop on Testing in XP, Alghero, Sardinia, Italy, May 27, 2002, Università di Cagliari, Free University of Bolzano-Bozen, BCI Italia, 2002.
  - [8] Sam Michiels, Dirk Walravens, Nico Janssens, and Pierre Verbaeten, *DiPSUnit: an Extension of the JUnit Test Framework for DiPS*, Report CW 333, Department of Computer Science, K.U.Leuven, Leuven, Belgium, February 2002.
  - [9] Christine Mingins and Chee Chan, *Building trust in third-party components using component wrappers in the .net frameworks*, Proceedings of the Fortieth International Confernece on Tools Pacific, Australian Computer Society, Inc., 2002, pp. 153–157.
  - [10] Judith Stafford and Kurt Wallnau, *Is third party certification necessary?*, Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering, Toronto, Canada, May 2001, pp. 13–17.
  - [11] Heffrey Voas, *A defensive approach to certifying cots software*, Tech. report, Reliable Software Technologies Corporation, August 1997.