Software Defect Masquerade Faults in Distributed Embedded Systems

Jennifer Morris, Philip Koopman
Carnegie Mellon University
{jenmorris, koopman}@cmu.edu

1. Introduction

Distributed embedded systems often consist of multiple nodes that communicate over a shared network. For such systems, dependable message delivery among nodes is crucial to overall system dependability. One threat to this dependable message delivery is a *software defect masquerade fault*, where a software defect causes one node or process to send a message as having come from another node or process. Unfortunately, many embedded system designs do not address this particular failure mode. This paper outlines what software defect masquerade faults are and why they are often ignored in current embedded systems. We also present preliminary research into methods to prevent them in embedded system design.

2. What is Software Defect Masquerading?

A masquerade is "in authentication, the pretence by an entity to be a different entity" [1]. In distributed embedded systems, masquerading usually occurs when one node or process sends a message across the embedded network that only another node or process is authorized to send. This usually occurs at the application level because resource constraints in many distributed embedded systems require the use of either a very simple operating system or none at all. In these systems the messages are constructed and sent directly by the application software.

In event-triggered networks that use a header field to identify the message sender, a node or process may masquerade by sending a message with a header field of another node or process. Time-triggered networks do not necessarily require headers if each node or process is allocated a particular transmission slot. In these networks, masquerading may occur if one node or process sends a message in the transmission slot of another node or process.

Although masquerading in distributed embedded systems can be intentional (i.e., a malicious attack from an intruder), it may also result from a defect in the system itself. In event-triggered systems, a software defect may cause a message with the wrong header field to be sent; in time-triggered systems a different software defect could cause a node to transmit during the wrong time slot. The result is a software defect masquerade fault.

3. Why is it Overlooked?

Traditionally, masquerading has been viewed as a malicious attack, rather than a fault tolerance problem. [2] looked at malicious masquerade attacks in open distributed agent-based systems and suggest strong cryptography as a method for preventing such attacks. Embedded systems, however, are typically closed (i.e., their networks are not accessible to outside intruders) and usually do not include security methods for preventing malicious attacks. In fact, most fault tolerance techniques used in embedded systems not only fail to prevent masquerading, but also assume fault models in which masquerade faults do not occur. For example, the Byzantine fault model assumes that the identity of each general is correct [3]. If software defects within the system itself cause masquerading that invalidates these assumptions, then the fault tolerance technqiques may or may not work.

Even if software defect masquerade faults are included in an embedded system design's fault model, the current methods for combating malicious masquerade attacks are often not practical to implement in most embedded systems. Authentication techniques such as strong digital signatures, which are designed to withstand malicious attacks and cryptanalysis, provide more security than is required to protect against inadvertent software defect masquerade faults, but require high processing and network overhead. Cost-focused embedded system designs often do not have enough resources for such strong cryptographic protection. For example, the eight bit microcontrollers that are prevalent in embedded systems are simply not designed to accommodate the complicated computational algorithms required to produce strong digital signatures.

4. How Can it be Prevented?

Although the assumption of a closed network is reasonable for most distributed embedded systems, nevertheless a dependable embedded system should provide protection against masquerading caused by software defects. The challenge is to find a method that will not add an unreasonable burden to product cost.

Many embedded systems developers attempt to mitigate the effects of software defects by dividing the embedded software into critical and non-critical components. Critical software is developed with an expensive, rigorous design process and is presumed to function correctly, whereas non-critical software is developed with a less expensive, less rigorous design process and is presumed to function incorrectly (for the purposes of safety cases). This method provides adequate protection against software defect masquerade faults in the critical components, provided that the critical and non-critical software are not transmitting messages over the same network. On a shared network, however, non-critical software masquerading of critical nodes and processes may still occur.

One way to prevent software defect masquerade faults on time-triggered networks is to use a bus guardian. Bus guardians are modules connected to each node on the network that prevent the node from sending messages outside of its designated transmission slot. Even if a node has a software defect that causes it to send messages in another node's time slot, the bus guardian will prevent the message from being sent at the wrong time. Masquerading may still occur if there is a defect in the bus guardian (incorrect synchronization, guarding the wrong time slot, etc.); however this method provides more protection than relying solely on the node itself to transmit correctly.

Another possible solution is to create a lightweight digital signature that uses relatively few system processing and bandwidth resources, yet is powerful enough to prevent software defect masquerade faults. Security research traditionally focuses on creating strong cryptographic protection; however, embedded systems, which have a more relaxed fault model but more rigid resource constraints, could benefit from lightweight cryptographic techniques for non-malicious fault scenarios.

One preliminary technique we have developed is to use a modified cyclic redundancy check (CRC) as a lightweight digital signature. Many distributed embedded system designs utilize an application-level CRC to verify message integrity. Although CRC's are useful at providing protection against random bit errors due to hardware malfunction or environmental interference, a CRC in its usual form has inadequate software defect masquerade coverage. In most implementations, the same CRC polynomial is used for multiple nodes on the network. For example, an embedded network for automotive or rail applications may use one or more CRC's for each message, but every message will use the same CRC polynomial(s) to generate the frame check sequence (FCS). If every node or process uses the same polynomial to generate the CRC, then the CRC calculated on a header field of one node or process will be correct whether or not it was actually generated by another node or process.

The CRC is converted into a lightweight digital signature by using a different CRC polynomial or a different seed value for each node to generate the CRC. The seed value is the initial value of the CRC register used in the calculation (all ones or all zeros in most CRC implementations). By setting this number to a distinct value for each node, the resulting CRC may be used as a lightweight digital signature, ensuring that software from any particular node cannot forge CRC values to masquerade on a different node. Further details on this method can be found in [4]

5. Ongoing Work

The ideas listed above are preliminary findings in ongoing research into the problem of software defect masquerade faults in distributed embedded systems. We are currently analyzing a variety of embedded network protocols, such as lightweight embedded IP, CAN, TTP, and FlexRay, to see where software defect masquerade faults may occur, as well as which attributes of the protocol help prevent them. In cases where the network fails to prevent software defect masquerade faults, we are exploring protection mechanisms that can be used at other system levels, such as the application level.

6. Acknowledgements

This work is supported in part by the General Motors Collaborative Research Laboratory at Carnegie Mellon University, Bombardier Transportation, and by the Pennsylvania Infrastructure Technology Alliance. The authors would also like to thank Bob DiSilvestro for his loyal support.

7. References

[1] Longley, D., Shain, M., and Caelli, W. *Information Security: Dictionary of Concepts, Standards and Terms*, Macmillan Publishers Ltd., Basingstoke, 1992.

[2] Minsky, Y., van Renesse, R., Schneider, F.B., and Stoller. S.D. "Cryptographic Support for Fault-Tolerant Distributed Computing". *Proc. 7th. ACM SIGOPS European Workshop*, 1996, pp. 109-114.

[3] Lamport, L., Shostak, R., and Pease, M. "The Byzantine General's Problem", *ACM Transactions of Programming Languages and Systems,4,3*, 1982, pp. 382-401.

[4] Morris, J. and Koopman, P. "Critical Message Integrity Over a Shared Network", *International Conference on Fieldbus Systems and their Applications*, July 2003.