2006 CCRTS

THE STATE OF THE ART AND THE STATE OF THE PRACTICE

Title: Ad-Hoc Networks and the
Mobile Application Security System (MASS)

Topics: Security of tactical MANETs,
Electronic Warfare Defense, Information Assurance

Author/POC: David Floyd

Bell Labs
Government Communications Lab
Room 15E-240
67 Whippany Road
Whippany, NJ 07981

Phone: 973.386.7099
Fax: 973.386.4821
Email: dfloyd@bell-labs.com

Abstract

Wireless tactical ad-hoc networks that use mobile applications to perform various tasks present a host of security concerns. By mobile applications, we mean applications that can support both moving nodes (e.g., receiver/transmitter on a Humvee) and migrating code (i.e., code that can transport itself via wireless link from one execution node to another). These networks, due to their wireless, open, and hostile operating environment, are more vulnerable to compromise by malicious applications and hosts than traditional networks that possess a clear line of defense. We are developing a solution to this problem that addresses critical aspects of security in ad-hoc mobile application networks. This approach involves preventing unauthorized modification of a mobile application, both by other applications and by hosts, and ensuring that mobile code is authentic and authorized. These capabilities constitute the Mobile Application Security System (MASS). The MASS applies effective, robust security to mobile application-based systems, wireless, and wired networks while minimizing overhead requirements. The MASS consists of innovative security techniques that provide distributed security solutions for mobile application networks.

Introduction

Innovative ideas typically bring new security challenges along with the promise of great improvements in technology. This is especially true in the field of Mobile Ad-hoc Networks (MANETs). While security is an important factor in any network, MANETs are open to many forms of injection attacks that are much less likely on traditional, wired networks. Additionally, MANETs do not typically employ firewalls, router Access Control Lists, Intrusion Detection Systems, or other network or gateway oriented defenses. Since traffic can enter the network at

any point, there is no clear line of defense in MANETs. When mobile applications are introduced (i.e., applications that can support both moving nodes and migrating code), the problem is compounded. How does one ensure that code that gains access to a new host is not malicious? How can application components know which other components are safe for interaction? How can the network prevent or, at worst, detect unauthorized modification of mobile code by a malicious host or program? To answer these questions, we suggest a solution that addresses critical aspects of security in ad-hoc mobile application networks, called the MASS. The MASS applies effective, robust security to mobile application-based systems, wireless, and wired networks while minimizing overhead requirements.

The MASS consists of an innovative layering of security techniques that provide a distributed security solution for mobile application networks. The various components of the MASS include a two-part distributed hashing and messaging approach to ensure that a mobile application has not been modified, a host-resident monitoring application to ensure an incoming mobile application is authentic and authorized before being allowed to install itself on the host, and signature variation to provide unpredictability and resistance to reverse engineering and malicious spoofing.

One example scenario for how the MASS works follows. Let us assume that the type of mobile application system under consideration is one composed of mobile software agents. For simplicity, imagine that the agents in a mobile agent system are all of a certain type, which we will call a color. At a given point in time, let us say that the agents in the system are all blue agents. A malefactor manages to capture a blue agent. He spends time reverse-engineering the

agent to understand how it works and then creates an agent that also looks blue to the other agents in the system, but which has been modified to perform some malicious activity. He then reintroduces this corrupt blue agent back into the system. The agents in the system are designed to check the color of any other agent they encounter. If the color of the encountered agent is not the same as their own, they refuse to interact with it. This would normally defeat less sophisticated modification and injection attacks. But because the bad agent has been cleverly designed to have the same color, all the other agents will accept it, and the system has been compromised. However, during the time the malefactor is working on the blue agent, what if all the other agents in the system automatically and unpredictably turn into some other color – say purple? The MASS provides this capability. So when the malefactor reintroduces the corrupt blue agent, the other (purple) agents will ignore it.

Our solution is distributed and capable of continuing to protect even when platforms and applications are removed from the system, corrupted, or destroyed (e.g., in a battlefield environment). No single security measure can provide the level of assurance required in a real-world environment. But the combination of the security mechanisms contained in the MASS reinforces the integrity and security of the mobile application system. The MASS layered security model creates a very small window for attack while requiring a prohibitively large amount of time to craft a successful attack. When utilizing the MASS and its innovative, dynamic, color-based security model, a hacker's chances of breaking into the mobile application network are negligible.

Technical Approach

Our security solution is intended to be independent of any particular mobile application framework. We do, however, envision that the deployed applications are given specific tasks and that several mobile applications for each task are deployed at any given time, with the number based on the required processing power or importance of each task. Therefore, at any moment in time there are several different types of applications in the system. Furthermore, all applications in the system at a particular moment in time belong to a specific generation. Every so often, the current generation is terminated and a new generation is created (as described below in "Application Time-To-Live").

Protecting Applications Against Unauthorized Modification

Our approach to this problem entails layers of defense, beginning with code obfuscation and signature variation, Time-To-Live (TTL) timers for the applications, and prevention of unauthorized modification via a hash system that works in conjunction with the other elements of the solution to help certify application integrity and detect code compromise.

Application Code Obfuscation

In order to modify an application's code to perform some malicious function, a malefactor would first have to reverse engineer the code to understand how it works. Therefore, the first layer of defense against unauthorized modification is code obfuscation. In this layer an obfuscator is run on the code to transform the program and make it very difficult to reverse engineer. There are a

host of code obfuscation techniques, depending upon the implementation language. Examples of these techniques are layout (e.g., scrambling identifiers), data (e.g., changing how data are stored in memory), preventative (e.g., exploiting known weaknesses of decompilers), and control (e.g., inserting “dead code” that will never be used). Of these methods, several of them will be used in our solution (preventative measures will not be used, due to their dependence on specific decompilers).

Application Signature Variation

After the application code has been obfuscated, a second layer of defense is applied via a technique to produce multiple code signatures. A program’s “signature” is the specific sequence (or order) of bits in its binary (executable) form. A value called a “hash” can be computed from any sequence of bits, which, with high probability, will be different if the sequence changes. Therefore, if a program’s signature changes, the program’s hash value also changes.

An example of this is code shuffling. This is a technique being developed at Bell Labs where data dependency between object code statements is determined and the object code is shuffled such that the dependency is maintained but the signature is transformed. Code shuffling creates a copy of a program that performs the same function as the original but whose signature is different.

Our approach uses a technique that generates multiple code signatures, such as code shuffling, to vary the hash values of applications between generations in a random, unpredictable manner. The added benefit of signature/hash variation is that, though improbable with present day

technology, it is theoretically possible for an attacker to acquire an application's hash and then construct a rogue application that possesses an identical hash value. The likelihood of such an attack will only increase as computing power improves. However, if an application's hash varies unpredictably, an attacker cannot know which hash to attempt to mimic. As a result, even in the highly unlikely event that an adversary successfully creates a rogue application, the time required to do so will severely limit that application's usefulness.

Application Time-To-Live

Another layer of defense is that applications periodically kill themselves in the system (based on a time-to-live variable assigned at creation) and are replaced by fresh applications whose object code has been obfuscated and shuffled. Alternatively, the arrival of the authenticated next generation of application could trigger the termination of the current generation. This avoids the denial of service attack where the application generator is rendered inoperative or its link is severed.

An application generator creates and deploys new applications. The generator then has no further contact with the applications. This enhances security because even if an application can be captured, successfully decompiled, and reengineered for malicious purposes, the time this will take will be far greater than the lifetime of that generation of applications. By the time the new malicious application is deployed, its brethren have died off and a new generation with completely different hashes are in the system.

Inter-Application Hash Comparison

When two applications need to interact, each must first compute the hash value of the other before accepting any directives or providing access to any of its resources. This could be, for example, a Secure Hash Algorithm hash, which generates 160-bit hashes, thus providing more security than other hashes that use 128-bit algorithms (e.g., MD5). [1] Comparing hashes is preferred over simple binary code comparison due to reliability and bandwidth restrictions in wireless networks. A SHA hash is only 160 bits long, which is easy to transport across the network, whereas a binary file can be arbitrarily large. Given the recent concerns over existing hash algorithms and their potential vulnerabilities [2] [3], alternative hashing algorithms are being explored, including the Very Smooth Hash (VSH) [4], which was jointly created by Bell Laboratories, Technische Universiteit Eindhoven, and Macquarie University.

When application A wishes to obtain a hash value for application B but the applications are on different hosts, the problem of obtaining a reliable hash arises. Application A cannot trust a value sent by application B, since it can be falsified. In order to solve this problem, the idea of helper agents is introduced. Helper agents are fully described in “Authorizing Applications.”

Verifying that another application’s hash is identical to one’s own suggests (but does not prove) that the other application has not been modified. We envision that the mobile application’s task will be small in nature and applications of the same type will communicate with each other. If one application detects that another application is not identical to it, the application sends a notification to its brethren (i.e., to members of its generation) that something is wrong. One of the two applications has an incorrect hash value. The applications then must decide which of the

two is the problematic one. This is accomplished by means of the following procedure: Some of the applications in the system will be given an additional task of monitoring other applications. These are called “sentinels.” In order to minimize the impact on system resources, it would be beneficial to have applications of the same type monitor each other. Here, an application’s type refers to the particular mission or task the application is performing.

Since there are already a number of applications in the system, this would prevent creation of extra applications simply for monitoring the applications that are doing the actual work. Also, by having the sentinel role performed by applications of a similar type, the amount of information needed and updated in each application would be minimal.

When the hash values of two applications do not agree, each application sends out an alert to the sentinels claiming that the other is corrupted. The sentinels must determine which one is “telling the truth” via a voting algorithm as follows: The sentinel applications receive a message that application *x* has computed a hash for application *y* that is incorrect. Each sentinel application calculates the hash value of application *y* and shares its observation (vote) with the other sentinels. If the application is determined to be faulty via a majority vote, it is ignored by the rest of its generation and terminated. This approach requires the sentinels to notify non-sentinels of their decision. If all applications are sentinels, then this notification is not necessary.

This prevents the corrupted application from spreading misinformation and disrupting the coordination process among legitimate applications. If the hash value of application *y* is determined to be correct, the same process is repeated for application *x*, to determine its integrity.

Application *x* cannot simply be assumed to be faulty and terminated, due to the possibility that a spoofing attack might be underway. The authenticity of the hash value, particularly when applications are exchanging hashes between remote hosts, is discussed in "Authorizing Applications."

Authorizing Applications

Our solution to the authorization issue with mobile applications involves platform-resident monitor applications, encrypted messaging between applications, and helper agents. A platform-resident monitor application dynamically determines whether mobile code should be allowed to execute on a new host. This application is installed on each host prior to deployment. It will have the ability to kill processes on the host, since it will be called upon to terminate rogue applications on the host after a consensus vote. At installation, the resident monitor application software is given an authorization key. After the host is added to the network, its resident application will only allow mobile applications with the proper key to run on the host. To assure the integrity of the resident application, each generation of mobile applications computes a hash of their resident application's software combined with a portion of their code and reports this to the other resident applications, which then use the distributed voting algorithm to remove any hosts with compromised resident applications from the network. It is important to note that, from an application-level perspective, the removal of the host involves ignoring any applications, resident applications, or messages originating from that host until the host has been cleared to re-enter the network.

Another method of authorization that the MASS uses is encrypted messaging. When each generation of applications is created, the applications are given a key for encrypting/decrypting messages. If an application receives a message that it cannot understand or decrypt, it discards the message and then checks the hash of the application that sent the unintelligible message. Since the key changes with every generation, applications from previous generations or modified applications reintroduced to the system will not be able to communicate and disrupt the current generation's operations.

Applications cannot be trusted to send their own hash values across the network to an application on a remote host, since a modified application could easily falsify the information. To counteract this problem, helper agents are employed. Helper agents are tiny programs spawned by an application that wishes to determine the hash value for a remotely located application. The helper agent contains only a minimal set of information (e.g., the hashing function) so as to reduce the bandwidth required to transport itself across the network. It travels to the remote host, performs a hash on the remote application, and returns to its parent application with the hash information. If the helper agent fails to return with the correct information, the parent application knows that there is a problem (i.e., the remote host and/or application is corrupt, the link is down, or the parent application itself is corrupt). In any event, the previously described distributed voting algorithm can be engaged in order to determine the nature of the problem. Once identified, the faulty/corrupt application can be destroyed. In the event of a broken link or a corrupt host, the host can be excluded from the network until an "all clear" can be established.

Conclusions

The MASS consists of an innovative layering of security techniques that provide a distributed security solution for mobile application networks. We provide remote hashing, code signature variation, application time-to-live, resident monitor applications, a mobile application generator, distributed application monitoring, code obfuscation, and hashing algorithms in a way that allows application-based systems to benefit from them while minimizing overhead requirements. In most existing application-based systems, security is not addressed or it creates bottlenecks and relies on a hierarchy. Our solution is distributed and capable of continuing to protect even when platforms and applications are removed from the system, corrupted, or destroyed. No single security measure can provide the level of assurance required in a real-world environment. But the combination of the security mechanisms outlined in this proposal reinforces the integrity and security of the application system. If a clever adversary defeats one measure, more than one alternative technique should not be defeated by the attack. The MASS layered security model creates a very small window for attack while requiring a prohibitively large amount of time to craft a successful attack. When utilizing the MASS and its innovative, dynamic, color-based security model, a hacker's chances of breaking into the mobile application network are negligible.

References

[1] Schneier, Bruce, Applied Cryptography, 2nd Ed., John Wiley & Sons, 1996.

[2] X. Wang, H. Yu, and Y. L. Yin. *Finding collisions in the full SHA*-1. In Crypto, 2005. 7

[3] Kaminsky, D. *MD5 To Be Considered Harmful Someday* 06-Dec-2004

[4] Contini, S., Lenstra, A.K., and Steinfeld, R. *VSH, an Efficient and Provable Collision Resistant Hash Function*. 2005