A SECURITY ARCHITECTURE FOR MEDICAL APPLICATION PLATFORMS

by

CARLOS SALAZAR

B.S., Kansas State University, Manhattan KS, 2012

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2014

Approved by:

Major Professor
Eugene Vasserman

Copyright

Abstract

The Medical Device Coordination Framework (MDCF) is an open source Medical Application Platform (MAP) that facilitates interoperability between heterogeneous medical devices. The MDCF is designed to be an open test bed for the conceptual architecture described by the Integrated Clinical Environment (ICE) interoperability standard. In contrast to existing medical device connectivity features that only provide data logging and display capabilities, a MAP such as the MDCF also allows medical devices to be controlled by apps.

MAPs are predicted to enable many improvements to health care, however they also create new risks to patient safety and privacy that need to be addressed. As a result, MAPs such as the MDCF and other ICE-like systems require the integration of security features. This thesis lays the groundwork for a comprehensive security architecture within the MDCF. Specifically, we address the need for access control, device certification, communication security, and device authentication.

We begin by describing a system for ensuring the trustworthiness of medical devices connecting to the MDCF. To demonstrate trustworthiness of a device, we use a chain of cryptographic certificates which uniquely identify that device and may also serve as non-forgeable proof of regulatory approval, safety testing, or compliance testing. Next, we cover the creation and integration of a pluggable, flexible authentication system into the MDCF, and evaluate the performance of proof-of-concept device authentication providers. We also discuss the design and implementation of a communication security system in the MDCF, which enables the creation and use of communication security providers which can provide data confidentiality, integrity, and authenticity. We conclude this work by presenting the requirements and a high level design for a Role-Based Access Control (RBAC) system within the MDCF.

Table of Contents

Table of Contents iv

List of Figures ix

List of Tables xi

Acknowledgements xi

Dedication xii

1 Introduction 1

2 Background 7

2.1 ICE 7

2.2 MDCF 9

2.2.1 Supervisor Components: 10

2.2.2 Network Controller Components: 11

2.2.3 Channel Abstraction 12

2.2.4 Connection State Machine 13

2.3 Related Work 16

3 Certificate Framework 17

3.1 Motivation 17

3.1.1 Trust Evidence 18

3.1.2 Public Key Infrastructure 19

3.2 Requirements 19
3.3 Design 22
3.3.1 Root Certificate 24
3.3.2 Manufacturer Certificate 24
3.3.3 Device Model Certificate 25
3.3.4 Device Instance Certificate 25
3.3.5 Trust Chain Creation 25
3.3.6 Authentication of Trust Chain 27
3.4 Certificate Authoring Tool 28

4 Authentication 30

4.1 Motivation 30
4.1.1 Requirements 31
4.1.2 Authentication Hooks 33
4.1.3 Authentication Providers 33
4.1.4 Robustness and Resource Allocation 34
4.2 Security Design 35
4.2.1 Device Authentication Hooks 35
4.2.2 Authentication Providers 37
4.3 Evaluation 39

5 Communication Security 43

5.1 Motivation 43
5.2 Requirements 44
5.3 Design 44
5.3.1 Hooks 44
5.3.2 Providers 46

5.4 Provider Implementation 48
5.4.1 NULL Provider 48
5.4.2 TLS Provider 48

6 Access Control 49

6.1 Introduction 49
6.1.1 Background 49
6.2 Requirements 53
6.2.1 System Overview 53
6.2.2 System Context 55
6.2.3 System Goals 55
6.2.4 Operational Concepts 55
6.2.5 External Entities 59
6.2.6 Requirements for the MDCF Access Control System 60
6.2.7 Requirements for Break Glass System 62
6.3 Design 63
6.3.1 Level 1: Minimal Implementation 64
6.3.2 Level 2: Customized Local Security Policy 65
6.3.3 Level 3: Multiple Users 66
6.3.4 Level 4: More Granularity 67

7 Conclusion and Future Work 69

7.1 Conclusion 69
7.2 Future Work 70

Bibliography 72

A Authentication Framework Performance Data 77

B Role Based Access Control System Use Cases **86**

B.1 Use Cases . . . 86

B.1.1 Normal Operation . . . 86

B.1.2 Break Glass Mode Operation . . . 87

B.1.3 User Log In . . . 88

B.1.4 User Log Out . . . 89

B.1.5 Device Connects (first time) . . . 90

B.1.6 Device Disconnects . . . 91

B.1.7 App Launch . . . 92

B.1.8 App Close . . . 93

B.1.9 Issue Command to Device . . . 94

B.1.10 Enter Break Glass Mode . . . 95

B.1.11 Exit Break Glass Mode . . . 96

B.1.12 (Break Glass) Device Connect (Unknown Device) . . . 97

B.1.13 (Break Glass) Device Disconnect . . . 97

B.1.14 (Break Glass) App Launch . . . 98

B.1.15 (Break Glass) App Close . . . 99

B.1.16 (Break Glass) Issue Device Command . . . 100

B.1.17 Create Role . . . 101

B.1.18 Delete Role . . . 102

B.1.19 Add Role To User . . . 103

B.1.20 Remove Role From User . . . 104

B.1.21 Add Permission to Role . . . 105

B.1.22 Remove Permission from Role . . . 106

B.2 Exception Cases . . . 107

B.2.1 User Log In (Failure) . . . 107

B.2.2 App Launch (Failure) 108

B.2.3 App Close (Failure) 109

B.2.4 Device Disconnects (Failure) 110

B.2.5 Issue Command To Device (Failure) 111

B.2.6 Exit Break Glass Mode (Failure) 112

B.3 External Entities 112

B.3.1 MDCF 113

B.3.2 DML Metadata 113

B.3.3 User management system 113

B.3.4 Supervisor App 114

B.3.5 Supervisor Console/Operator Interface 114

B.4 Requirements 115

B.4.1 RBAC System Function 115

B.4.2 Break Glass System Function 118

List of Figures

1.1 Security-related threats to MAPs 3

1.2 Network based attacks on MAPs 4

1.3 High level MAP security goals 5

2.1 The primary components of the ICE architecture. External interface, and patient connected to devices, are omitted. 8

2.2 MDCF components grouped by their logical ICE architecture role. 10

2.3 The MDCF Channel Abstraction. 12

2.4 MDCF Connection State machine, with the outline denoting portions relevant to connection-time authentication. 14

3.1 FDA's sample Unique Device Identifier label for medical devices. 18

3.2 ICE certificate chain. Solid lines indicate signing relationships, i.e. the private key corresponding to the arrow origin is used to sign the certificate to which the arrow points. The dotted line represents a certificate extension used to accommodate the additional signature of the Certifying Authority on a certificate. 21

3.3 MDCF Certificate Authoring Tool – a GUI to streamline certificate creation for the ICE certificate chain. Pictured is the tab used for creating device model signing requests. 29

4.1 MDCF components grouped by their logical ICE architecture role and showing primary hook locations (circles a,b,c). 32

4.2 Illustration of the authentication process using the `NULL` provider. 37

4.3 MDCF resource usage with 340 virtual devices running on a different host. . 41

5.1 MDCF components grouped by their logical ICE architecture role and showing primary hook locations (circles a,b,c). 45

5.2 MDCF channel security providers. 46

6.1 Flat RBAC[1] 50

6.2 Pictured is a context diagram for the RBAC system. 54

List of Tables

3.1 Information used to judge the trustworthiness of a device. 22

4.1 Authentication provider (device- and MDCF-side), complexity measured using lines of code (LOC), and complexity increase from the NULL provider. NULL is little more than the common infrastructure/scaffolding. The "Increase over NULL" column is therefore a more accurate representation of the code complexity increase of new authentication modules. 40

6.1 High-level goals(G) of the RBAC system 55

6.2 Summary of use cases 56

6.3 Summary of exception cases 57

6.4 Actors in the RBAC system 57

Acknowledgments

First and foremost, I wish to thank my adviser, Dr. Eugene Vasserman for all of his guidance and encouragement over the past two years. I also want to thank Dr. John Hatcliff for the advice and suggestions he has given me and for agreeing to serve on my committee. I'd like to thank Dr. Xinming Ou for serving on my committee, and also for his mentorship through the Cyber Defense Club. Furthermore, I want to thank Dr. Dan Andresen for originally giving me the opportunity to get involved with research and for his work with Beocat. Finally, I want to thank my friends and family for the support and encouragement you have given me through the years.

Dedication

For Amy Hickson and J.E Townley.

Chapter 1

Introduction

Medical Application Platforms (MAPs)[2] such as the Integrated Clinical Environment (ICE) are predicted to enable improvements to health care through the integration and coordination of heterogeneous medical devices. A MAP provides a common interface to which devices from different vendors may be connected, and on top of which, medical "apps" may be developed. Apps on a MAP may receive and display data from one or more devices also connected to that MAP, furthermore a MAP enables the configuration and control of devices by apps.

The vision of medical devices in MAPs stands in a stark contrast to the status quo. Current medical devices are developed in a monolithic, stand-alone manner. They offer little to nothing in terms of interoperability features. When connectivity is present it is limited to a unidirectional transmission of device data for logging or display purposes.[3] Furthermore, this connectivity is further restricted by a lack of interface compatibility with components from other manufacturers. The result of this incompatibility is the creation of vertically integrated systems culminating in a lack of technological diversification.

The Integrated Clinical Environment (ICE) is a ASTM Standard describing a conceptual architecture of a MAP. As with any MAP, ICE is intended to integrate different medical devices originating from different manufacturers and allow apps to control devices and display their data. ICE components and systems are meant to function in a plug-and-play manner. The integration of ICE components is designed to be conducted dynamically, on demand by non-technical users (i.e. clinicians) at the point of care facility.

Consequently, ICE requires a new component-wise regulatory approach. This differs from existing regulatory practices for medical devices and for other interoperable systems that rely on a known a-priori configuration assembled by a skilled technical integrator. Without the new approach to regulation for MAPs, building an ICE-compliant system will not be feasible.

Although MAPs are thought to be beneficial to patient safety,[4,5] they also open the doors to security-related threats to patient safety and privacy. In Figure 1.1 we have identified some threats which motivate the need for security features in MAPs. A necessary, but not sufficient part of mitigating these threats, is protecting against specific network based attacks (see 1.2).

These attacks are by no means unique to MAPs — they are well known in the context of other networked and distributed systems. In this regard, the greatest difference between this and most other systems is that many classes of attacks (e.g. denial of service, man-in-the-middle) can directly cause harm to a patient on a MAP. In essence, it is impossible to fully separate the concepts of safety and security when dealing with MAPs. Aside from threats to privacy and accountability, **a security threat in a MAP should be considered a safety threat**.

Aside from network-based attacks, components that do not meet regulatory, safety, or other compatibility standards have an increased potential to compromise patient safety through design or implementation problems. These components may also contain back-doors or software vulnerabilities that constitute a security threat if they are allowed to be integrated into an interoperable medical system. Furthermore, a MAP also has the potential to expose sensitive patient data or threaten patient health if someone other than an authorized clinician is able to access the devices or apps associated with a patient through

1. **Unauthorized Supervisor Console Access:** An individual other than an authorized user (i.e. clinician) who is able to view or gain control of the apps and management/administrative controls of a MAP will be able to control devices and view sensitive patient data.

2. **Unauthorized or malicious control of a device:** Devices that perform some actuation on a patient may be leveraged to directly harm to patients through an improper actuation, or through the prevention of some necessary actuation. Misconfiguration of devices might also interfere with delivery of care.

3. **Damage to a device:** Damage or destruction of a device may prevent necessary actuations on a patient, or prevent the collection of physiological data, potentially causing harm the patient.

4. **Unauthorized access to patient data:** Violation of patient privacy and laws such as HIPAA.

5. **Destruction of patient data:** Destruction of patient data may interfere with proper delivery of care, and may be used to violate accountability.

6. **Disruption of system functionality:** Disruption of networking or platform functionality can interfere with care delivery.

7. **App/Device data tampering:** Data directly displayed to users through apps, or that is used by apps in an intermediate sense to drive app logic can harm patients by misleading clinicians or causing apps to respond inappropriately.

8. **Log data tampering:** Alteration or destruction of data destined for or contained by the logging system can be used to violate accountability.

Figure 1.1: *Security-related threats to MAPs*

normal user interfaces.

Figure 1.3 contains a set of high-level security goals that we have created for MAPs. These goals are derived from the security properties and requirements described by Vasserman et al.[6]

The work presented in this thesis is intended to be a step towards the creation of a comprehensive security architecture for the Medical Device Coordination Framework (MDCF). The MDCF is a functioning Medical Application Platform, which has been built to serve as a working implementation of an ICE-like system. In order to meet our security goals (1.3,

1. **Denial of Service (DoS):** As in any network connected system, denial of service attacks are possible. For example, The underlying communication infrastructure could be flooded with messages. The ability to connect a device might be impeded by a flood of connection requests.

2. **Resource Consumption:** A resource consumption attack is a subset of denial of service attacks. Many types of DoS rely on the attacker having more resources than the target, but in resource consumption attacks the amount of resources an attacker uses to attack a system is disproportionately smaller than the quantity of the target's resources consumed in the attack.

3. **Eavesdropping** An attacker is able to view and store sensitive plaintext data.

4. **Message Replay:** An attacker is able to record some message (data transmission or a command), when the copied message is sent again, it is treated as a valid message.

5. **Message Forgery:** An attacker is able to generate and insert messages/data into the system as they please.

6. **Man-in-the-middle:** Using a combination of other attacks, the attacker is able to insert themselves in communications between components of a MAP, impersonating components in a way that allows them to quietly control the system and potentially cause serious harm to a patient

Figure 1.2: *Network based attacks on MAPs*

we have explored device certification, communication security, device authentication, and access control for the MDCF.

A certification system is necessary to determine the provenance of components to be integrated into the MDCF. This system is directly required for meeting our goal (1.3) of ensuring that components integrated into a MAP have necessary regulatory approval. Models of compatible devices will need to be approved by a regulatory/certifying authority. The manufacturers of devices must also be approved by this authority, and each individual device will have a unique identifier assigned to it. We need a means of reliably enforcing that all necessary certification steps have been taken for each device before it is allowed to be integrated into the MDCF or other ICE-like system. Thus, one of the contributions of this work is to show a straight-forward mapping of regulatory policy to a verifiable cryptographic

1. Prevent attacks and unauthorized access to the system that could directly or indirectly cause harm to patients
2. Prevent theft or other exposure of protected health information
3. Ensure components integrated into the system have appropriate legal, safety, regulatory approval
4. Ensure that security features are highly usable (security does not interfere with delivery of medical care)

Figure 1.3: *High level MAP security goals*

representation/indication of component-wise regulatory approval.[3] We accomplish this using chains of certificates which are packaged with every ICE compatible device, which are used to authenticate devices. During the authentication process for a device, if any portion of the chain of certificates does not match the rest of the chain, or if the certifying authority is not recognized, the device will not be allowed to connect. Otherwise, if a device's certificate chain is found to be valid, it will be allowed to continue on to the other steps in the connection process.

In order to address threats (1.1) directly stemming from network based attacks (1.2), communication security and device authentication features must be incorporated into the MDCF. We believe that these features should be implemented modularly, enabling medical device vendors to implement whichever features they deem necessary. Ideally, pre-built standard communication security and authentication modules will be available from a certification body or third-party software developers.[2] We intend to maximize compatibility by offering numerous security provider choices, and accommodate new technologies as they emerge by making our system extensible. Our modular implementation approach is similar to that of Java or OpenSSL security services,[7,8] instantiated by name rather than a function call to a specific method.[9] By offering pre-made security providers, we decrease the workload of medical device developers by making it unnecessary to write channel security and authentication modules from scratch.

Access control, together with an user authentication system, is necessary for preventing threats (Figure 1.1) to patient safety and privacy, particularly those stemming from unauthorized use of clinician user interfaces. Given the nature of a clinical setting, we believe that an access control system based on Role-Based Access Control (RBAC)[1] is an appropriate solution. RBAC is designed to allow administrators to define access control policies that fit the roles and workflows of their organization.[10] Our exploration of RBAC is in the form of a requirements specification and a high level design for an RBAC system in the MDCF.

Following this introduction is a background chapter. In the background we describe the context of this work in terms of previous and related efforts on device interoperability. This includes some discussion on the Integrated Clinical Environment (ICE), and how it relates to the MDCF. We describe the architecture of the MDCF, and the purpose of the various components in it. This also includes a description of the MDCF Channel Abstraction, and the Connection Mode State Machine. After the background, we describe the device certification system we have devised for ensuring the trustworthiness of medical devices connecting to MAPs. Next, we discuss the design and implementation of a communication security system in the MDCF, which enables the creation and use of communication security providers which can provide data confidentiality, integrity, and authenticity. We also cover the creation and integration of a pluggable, flexible authentication system into the MDCF, and evaluate the performance of proof-of-concept device authentication providers. Finally, we present the requirements and discuss design choices for a Role-Based Access Control (RBAC) system within the MDCF.

Chapter 2

Background

2.1 ICE

The Integrated Clinical Environment (ICE)[11] is ASTM standard F2761-2009[12] for a medical application platform – a safety- and security-critical high-assurance middleware for inter-component communication.[2] It is intended to serve as a universal standard for medical device integration. The clinical functionality of the system is provided by medical devices as well as automated clinical work flows ("apps") running on an application host. ICE apps can both receive data from devices and control them. This allows developers to create apps which can perform transformations on the data, synthesize data from multiple devices, and/or perform closed-loop control.[13] A closed-loop control app would describe the data required (e.g., types and frequency of sensor readings), the nature of its service in terms of device control (actuation), as well as explicit safety goals. Another possible application is a "smart alarm"[13] which is triggered by a combination of physiological signals from multiple devices. This sort of alarm may suffer from less false positives than traditional alarms. ICE apps allow for richer, more powerful ways to view and analyze physiological data than would be present directly on a device. This also means that some devices such as pulse oximeters can function more as sensors without a user interface, instead of being full

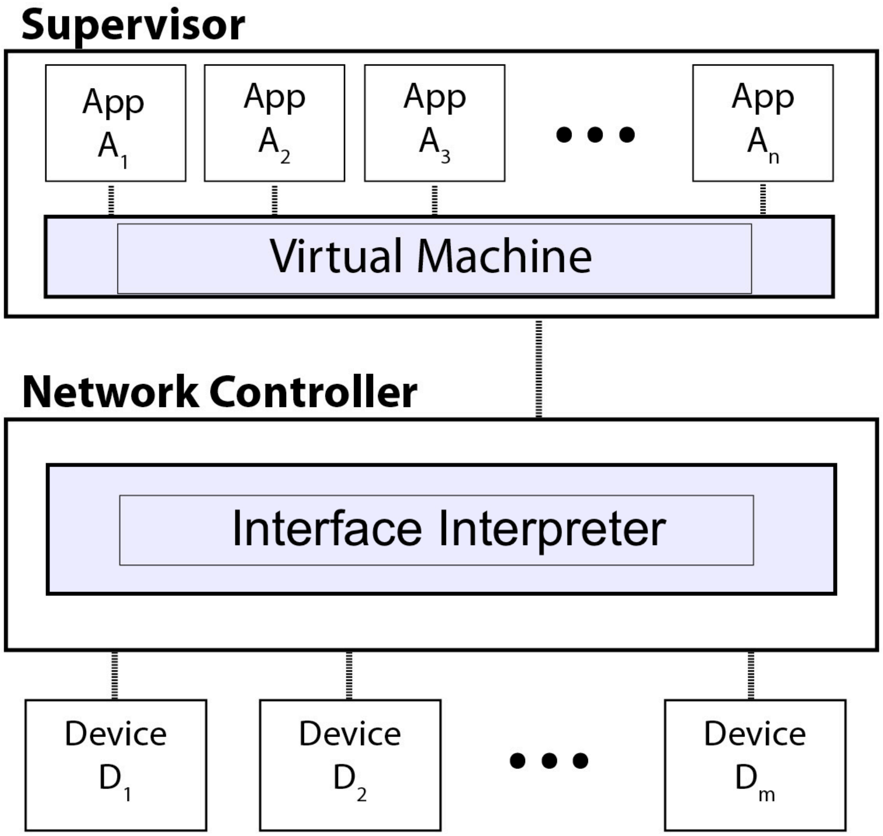


Figure 2.1: *The primary components of the ICE architecture. External interface, and patient connected to devices, are omitted.*

standalone devices.

An important feature of ICE is that it suggests a new approach to medical device regulation. In the current regulatory philosophy, a system like ICE would likely require regulatory approval for each possible combination of connected medical devices, because each configuration could be considered a different medical "device".[2] Consequently, ICE suggests a component-wise regulatory approach, wherein various standards and interfaces are defined for ICE, and if a device is found to conform to these standards, we can infer that it is safe to connect to the ICE system. In this approach, the app manufacturer is responsible for making system safety claims, and we rely on the ICE system (enforced by specialized middleware) to prevent devices from adversely impacting each other's functioning. Thus, if each individual device or app has an ICE compatible interface that passes safety testing and the ICE middleware does not allow devices to interfere with each other, then any[1] set of connected ICE compliant devices and apps should still result in a safe system.

In ICE architectures, devices are connected to a component called the Network Controller. This component can be considered the network abstraction: it facilitates communication between devices and applications (automated medical work flows) running on the Supervisor. Figure 2.1 illustrates the basic architecture of ICE.

The ICE Supervisor encompasses the "front end", user-facing component of ICE, and hosts apps in isolated environments and guarantees run-time resources like RAM and CPU time. In the ICE architecture, apps are programs that can display patient data as well provide control over devices which support it. The Supervisor also includes built in tools for launching apps and managing devices.

The ICE Network Controller facilitates communication between Supervisor apps and medical devices. All ICE communication takes place through messages sent over virtual channels maintained by the Network Controller (NC). Each channel is specific to a device-app pair. Whenever a new device connects to an ICE system, it is the Network Controller that "discovers" the new device and performs the connection/handshaking and authentication protocols.

2.2 MDCF

The Medical Device Coordination Framework (MDCF)[14] is an open source medical application platform, developed jointly by Kansas State University and the University of Pennsylvania.[13] It is intended to serve as a sandbox that allows researchers, regulators, and

[1] There will be limits on how many devices and apps can be simultaneously connected based on computational resources such as CPU, memory, storage, and bandwidth. Ideally any combination of compliant apps and devices that fall within our resource constraints should result in a safe, functional system.

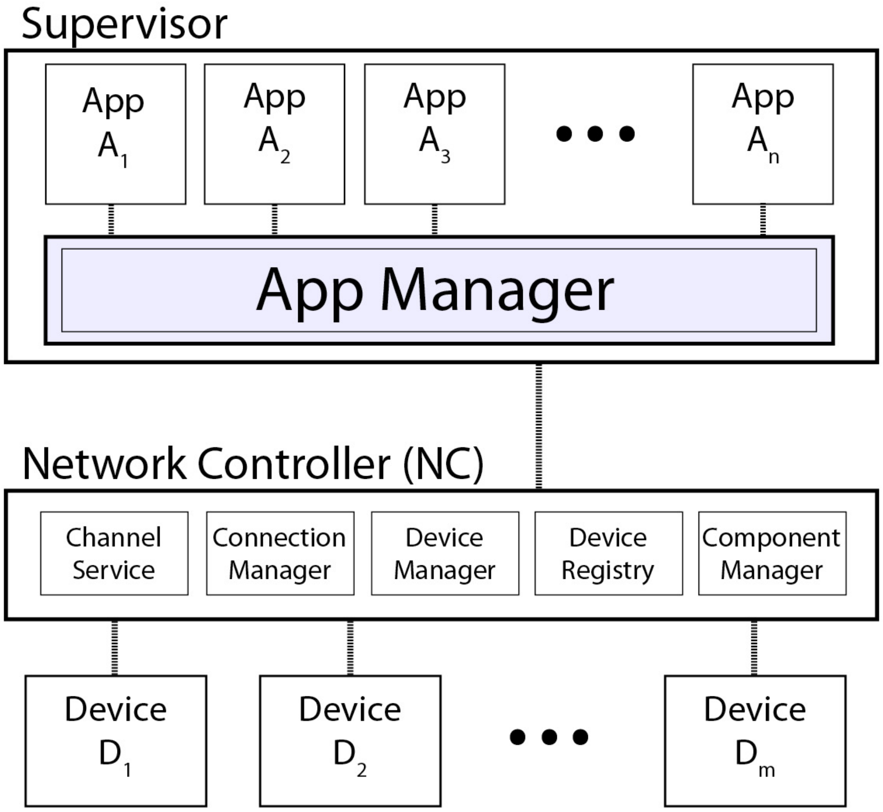


Figure 2.2: *MDCF components grouped by their logical ICE architecture role.*

medical device vendors to experiment with a working ICE-like system. Figure 2.2 shows the organization of MDCF components with respect to the ICE architecture described in the previous section. They are described in detail below.

2.2.1 Supervisor Components:

The App Manager: manages the life-cycle of apps; meaning that it starts and stops the execution of apps, provides isolation and service guarantees, monitors and resolves (or notifies clinicians of) "clinically important" (e.g. medically adverse) interactions or architectural interactions (two apps trying to get exclusive control of one device).

The Clinician Service: provides an interface for selecting, instantiating, and configuring Supervisor apps for use with a clinician console GUI. New apps can be started and running ones can be configured. Appropriate user authentication/login will be required.

The Administration Service: provides controls for managing and installing Supervisor apps and components. Appropriate user authentication/login will be required. This service should not need to reconfigure running applications, and should be prevented from doing so by the App Manager.

2.2.2 Network Controller Components:

The Channel Service: provides interfaces between middleware platforms and the rest of the MDCF. It contains interfaces for the messaging server (e.g. OpenJMS, ActiveMQ),[15] message senders, message receivers and message listeners. It is partially responsible for inter-app and inter-device data isolation and performance guarantees. It houses the code for all authentication providers, as well as all interfaces and factories related to authentication providers (Figure 2.2(a)).

The Connection Manager: manages connections with devices and the creation and destruction of channels through direct interaction with the message provider. The Connection Manager is directly involved with device authentication. It also contains the main hooks for the Network Controller authentication providers (Figure 2.2(b)).

The Device Manager: sets the status of a device as connected or disconnected, sends commands to device to start or stop publishing, and configures devices for use with specific apps.

The Device Registry: stores and retrieves information about devices from a database. For each device, it stores and provides access to information such as its type, name, metadata, and active apps associated with it. We augment this data structure to store security metadata, such as active encryption keys for device private channels.

The Component Manager: manages MDCF app components, and works in a way analogous to the Device Registry. It is used to store and retrieve information about app components.

2.2.3 Channel Abstraction

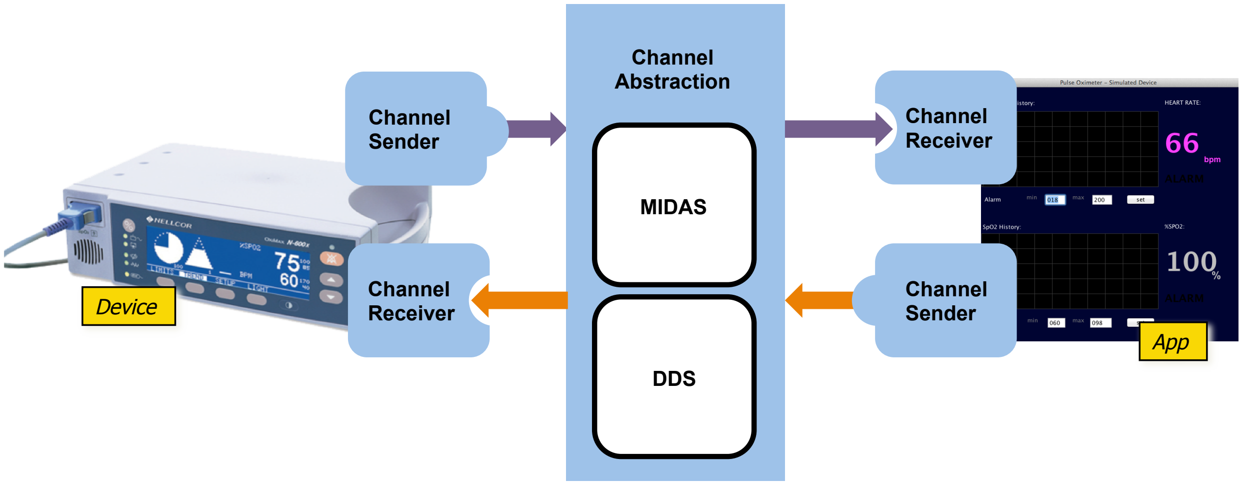


Figure 2.3: *The MDCF Channel Abstraction.*

A critical element of the architecture of the MDCF is the channel abstraction (figure 2.3). The channel abstraction allows the MDCF to run on top of different middleware implementations, allowing them to be treated as a message-oriented publish/subscribe middleware platform. Pictured here are MIDAS and DDS, the two supported middleware systems at the time of writing. However, the MDCF has in the past also supported ActiveMQ and OpenJMS.

The channel abstraction relies on a notion of channels (also sometimes called topics). Channels are used to isolate streams of messages such that messages in a stream can only be sent or received by components that register a sender or a receiver object for that channel. Thus a message is only ever distributed to components that have a receiver subscribed to the channel the message was published on.

2.2.4 Connection State Machine

The implementation details of the device connection protocol (using a state machine), shown in Figure 2.4, are particularly relevant for device authentication. Each state is implemented as a separate Java class. It is within these classes that the messages in the connection process are sent and received. Effectively, two connection state machines exist for each device; the device and the Network Controller each maintain their own separate views of this state machine. These different views of the connection state machines are utilized to ensure that all of the steps in connection process are executed in the appropriate order. (In the text, **we refer to the Network Controller view** of the state machine, unless stated otherwise.) Each state machine is associated with a single object that can be used to access or modify the current state. These classes are also used to store any information that needs to be accessed by more than one state. The states most relevant to device authentication are:

DISCONNECTED: The initial state. The device sends the AUTH message during the DISCONNECTED state. Upon reception of the AUTH message, the Network Controller initializes its view of the connection state machine and moves into the AUTHENTICATING state.

AUTHENTICATING: Upon receipt of the AUTH message, the Network Controller allocates and connects to private channels for the device and sends the channel information to the device in an AUTH_ACCEPTED message. The device connects to the channels, after this point the rest of the messages used for connection are sent across these private channels. Note that “private” is used here not to denote confidentiality, but rather than these channels are logically dedicated to communication with a specific device (as opposed to the public “atrium” channel).

AUTHENTICATED: The device has been successfully authenticated. It sends an INTERFACE message to test the private channels before progressing into the ASSOCIATING state. (The INTERFACE message is a confirmation that the private channels set up at the

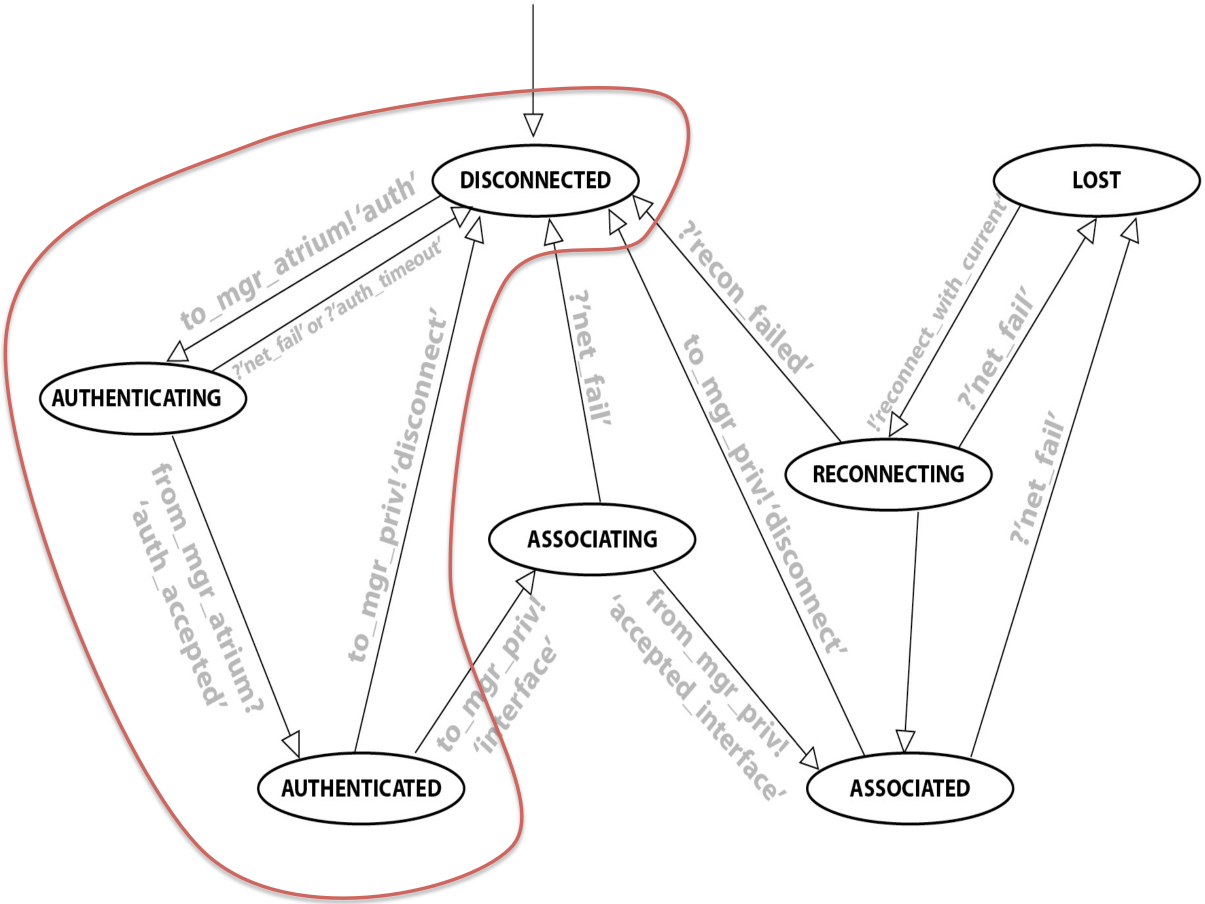


Figure 2.4: *MDCF Connection State machine, with the outline denoting portions relevant to connection-time authentication.*

end of the AUTHENTICATING state are working. The content of the message is a fixed string.) Although we routinely refer to the device authenticating to the Network Controller, it is trivial to extend the protocol to support mutual authentication.

ASSOCIATING: Upon receipt of the INTERFACE message, the Network Controller creates heartbeat and acknowledgment channels. The device periodically publishes heartbeat messages on this channel, enabling the Network Controller to detect an unexpected device disconnection (if too many heartbeat messages are missing). The Network Controller communicates these private heartbeat channels to the device and then transitions to the ASSOCIATED state.

ASSOCIATED: The device is fully connected in this state. The device will remain in this state unless it ceases to be connected and transitions to either the LOST or DISCONNECTED state.

LOST: When too many device heartbeats have been lost, the Network Controller places the device in the LOST state. It must then attempt to reconnect, transitioning into the RECONNECTING state. (If the device state machine is not in the LOST state, and it successfully communicated with the Network Controller, it will be explicitly told to reconnect.)

RECONNECTING: A device in this state is attempting to reconnect. If reconnection is successful, the device it returns to the ASSOCIATED state. Otherwise, the device transitions to the DISCONNECTED state. The Network Controller remains in the RECONNECTING state for a fixed amount of time, or until the device successfully reconnects.

In practice, to minimize resource usage and protect against resource DoS attacks, the state machine object used for a connection by the Network Controller (in the transition from DISCONNECTED to AUTHENTICATING) is taken from a pool of pre-allocated objects. When the Network Controller "destroys" a state machine, it is returned to the object pool. Note that this prevents devices from connecting when the pool is exhausted (either due to a large number of connected devices or an active attack). This is by design: devices that authenticate successfully will eventually connect, and devices which cannot authenticate but are performing the attack cannot cause more objects to be allocated. Although malicious devices may keep the pool drained, an honest client will eventually (probabilistically) succeed in sending an AUTH message to the MDCF through the flood of adversarial messages, thus reserving a provider. Authenticated connections are long-term, so honest devices need only succeed once. When under attack, the time needed for an honest device to connect may be arbitrarily long, but in practice would be bounded with high probability.[16]

2.3 Related Work

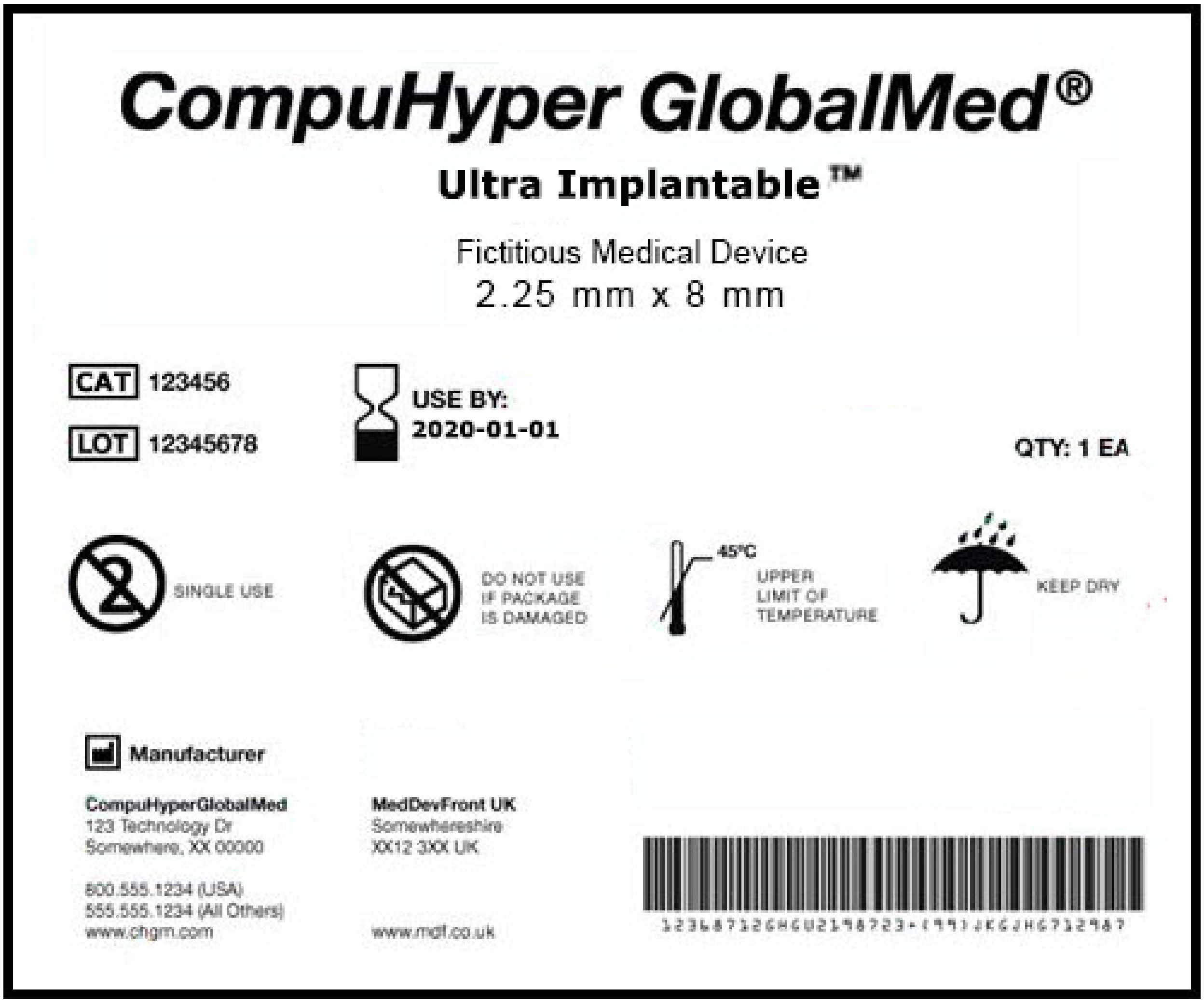
Over the past several years, several papers have been published specifically on MAP/ICE security. Hatcliff et al.[3] establish the need for certificate based trust framework in ICE. Venkatabramanian[17][18] has highlighted the need for securing interoperable medical devices and described risks, attack vectors and security requirements for them. Venkatabramanian et al.[19] present an attack model for ICE-like systems. Vasserman et al.[20] present models for failures and the consequences of failures for components of ICE-like systems. Vasserman and Hatcliff[6] describe security challenges and requirements for MAPs. Kune et al.[21] discussed ICE security requirements at different levels of the OSI stack. Taylor et al.[22] used attack graphs to evaluate threats to interoperable medical devices and discussed possible mitigations to those threats. The chapters in this thesis on authentication and communication security cover material previously published by Salazar and Vasserman.[9]

Chapter 3

Certificate Framework

3.1 Motivation

In order to safely use a dynamically-constructed system of medical devices and apps, we must be able to determine the provenance of the components. Models of ICE compliant devices will need to be approved by a regulatory/certifying authority. The manufacturers of these devices must also be recognized by the FDA and individually identified, e.g., with an FDA-assigned unique number. We need a means of reliably enforcing that all necessary certification steps have been taken for each device before it is allowed to be integrated into an ICE system. In this paper, we show a straight-forward mapping of regulatory policy to a verifiable cryptographic representation/indication of component-wise regulatory approval.[3] We accomplish this using chains of certificates which are packaged with every ICE compatible device, which are used to authenticate devices. During the authentication process for a device, if any portion of the chain of certificates does not match the rest of the chain, or if the certifying authority is not recognized, the device will not be allowed to connect. Otherwise, if a device's certificate chain is found to be valid, it will be allowed to continue on to the other steps in the connection process.



CompuHyper GlobalMed®

Ultra Implantable™

Fictitious Medical Device
2.25 mm x 8 mm

CAT 123456

LOT 12345678

USE BY:
2020-01-01

QTY: 1 EA

SINGLE USE

DO NOT USE
IF PACKAGE
IS DAMAGED

45°C
UPPER
LIMIT OF
TEMPERATURE

KEEP DRY

Manufacturer

CompuHyperGlobalMed
123 Technology Dr
Somewhere, XX 00000

800.555.1234 (USA)
555.555.1234 (All Others)
www.chgm.com

MedDevFront UK
Somewhereshire
XX12 3XX UK

www.mdf.co.uk

123687126HGU2198723-(99)JKGJHG712987

Figure 3.1: *FDA's sample Unique Device Identifier label for medical devices.*

3.1.1 Trust Evidence

Existing medical devices have human-readable labels that identify the device, display manufacturer recognition from regulatory agencies, and labels that attest to safety/compliance testing for the product. In Figure 3.1, we see a sample human-readable label from a recent FDA guidance on device and component identification.[23]

To ensure appropriate trust levels, ICE needs non-forgeable and machine-readable labels that provide information about the manufacturer, product, and confirmation of compliance/safety testing and regulatory review. This machine readable information is not meant to replace the physical label – both should be present on an ICE compliant device.

3.1.2 Public Key Infrastructure

Our certificate framework is based on a Public Key Infrastructure (PKI).[24] Public key cryptography provides an efficient, distributed way to prove the identity of an entity without requiring an always-on Internet connection, e.g., to query an FDA database. Each entity has a certificate which contains information about that entity, particularly their name and public key. Each certificate is signed using the private key corresponding to some other entity. In PKI, there is a **root of trust** who serves as the starting point in a **chain of trust**. All other certificates are signed by either the root or an intermediate authority. Intermediate certificates are similar the root certificate except that they are signed by the root. Finally, there are end-entity certificates that are intended only to identify an entity, and may not be used to issue other certificates. Certificates may also carry additional information about an entity, beyond what is included in a standard certificate. In the X.509 PKI standard, this additional information is contained within extra fields appended to the end of a certificate, known as certificate extensions. For our proof-of-concept implementation, we use X.509v3 certificates with a custom extension.

3.2 Requirements

A high-assurance compositional system calls for an automatic way to verify the provenance and regulatory/third-party approval status of devices and apps before they are allowed to authenticate.[3] A trust architecture should complement the proposed component-wise regulatory approach for ICE. In this section we lay out the specific requirements that a trust architecture for ICE should satisfy.

We assume that a device is trustworthy if it has been shown to be ICE compatible, has received regulatory approval (by the FDA or delegated third-party agency in the United

States, or any appropriate agency for a particular geographical region) and has optionally received additional third-party safety and/or compliance certification. Thus, our main concern is reliably conveying this information about a device to ICE. A specific list of information needed to determine trustworthiness for a device is in Table 3.1.

It may be possible to create malicious software or devices that interfere with the safe functioning of the ICE system as a whole, just as one may introduce malware into even tightly controlled ecosystems, such as Apple's iOS.[25] A malicious component could exfiltrate sensitive patient information or interfere with the functionality of other system components. A malicious device might even harm patients directly, while a malicious app could do so indirectly by reporting inaccurate information or sending commands to devices.

It is out of scope for this work to examine, in detail, how we might prevent malicious entities from infiltrating medical systems, or causing harm if and when they do. We acknowledge that compatibility, regulatory, and safety checks may be insufficient to catch sophisticated attackers. Nonetheless, we wish to prevent misrepresentation of capability and misidentification of components or manufacturers. We focus on preventing the forgery of compatibility and safety evidence. The ICE trust framework must be designed so that devices and apps with forged, stolen, or otherwise invalid credentials will not be trusted by an ICE system. The transmission and verification of trust information about a device needs to be done quickly, in an automatic fashion which involves minimal to no interaction with clinicians. Such a system should also be created with an eye towards ease of use for regulators/third-party certifiers, manufacturers, and system administrators at point of care facilities. We believe that these requirements point towards use of a light-weight PKI based on X.509 certificates.

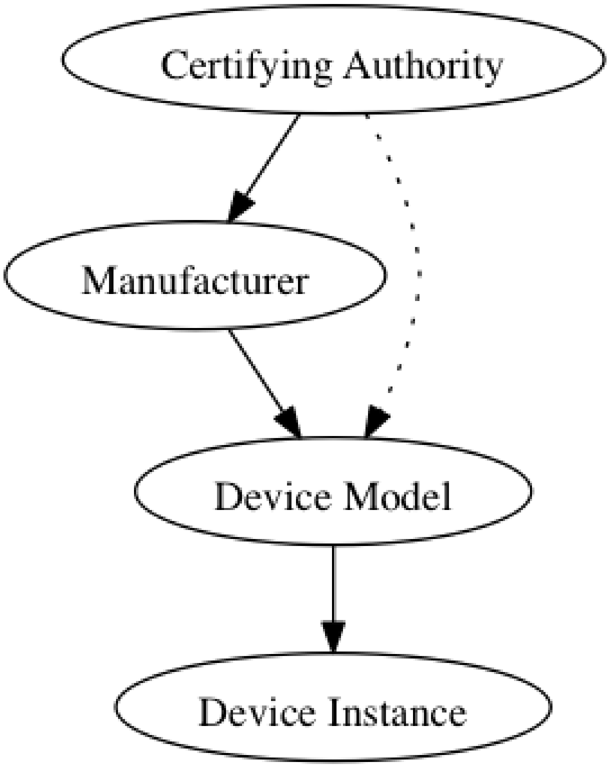


Figure 3.2: *ICE certificate chain. Solid lines indicate signing relationships, i.e. the private key corresponding to the arrow origin is used to sign the certificate to which the arrow points. The dotted line represents a certificate extension used to accommodate the additional signature of the Certifying Authority on a certificate.*

1. **Device global unique identifier (GUID)**
2. **Identity of the device manufacturer**
3. **Specific device model information**
   (e.g., model name, firmware revision)
4. **Evidence of ICE compatibility**
5. **Evidence of regulatory/safety compliance**

Table 3.1: *Information used to judge the trustworthiness of a device.*

3.3 Design

Trust evidence for an ICE compliant medical device is represented as a chain of X.509v3 certificates. This chain is sent by the device to the ICE Network Controller during the connection process. If the certificates are valid, the device is considered trustworthy and will be allowed to proceed in the connection process, otherwise the device will be prevented from connecting. We define valid certificates as those which can be traced back to a root of trust whose certificate is stored by the ICE NC, are within their valid date range as listed on the certificate, and if applicable, have an issuer field that matches the subject field of the certificate corresponding to the private key used to sign the certificate.

This chain of certificates contains all of the information listed in Table 3.1. Specifically, a device's unique identifier (GUID, Table 3.1-1) is stored in its device instance certificate. The manufacturer of a device (Table 3.1-2) is identified by the manufacturer certificate in the certificate chain. Device model information (Table 3.1-3) is contained within the device model certificate. Evidence of ICE compatibility and regulatory/safety compliance (Table 3.1-4 and 5) is represented by the signature from the Certifying Authority on the device model certificate. The Certifying Authority will only sign device model certificates which have met regulatory requirements.

Use of the PKI scheme described in this section satisfies our requirement of providing a non-forgeable means of obtaining necessary trust information as long as the underlying public key and hashing algorithms are valid and applied correctly. (It is computationally infeasible to find the private key matching a public key using a sufficiently large key size.)[26] From the certificate chain, the verifier should be convinced (as long as a signature verifies correctly) that e.g., the device which presents the chain was developed by the claimed manufacturer and cleared by the Certifying Authority.[1] Thus, as long as the various entities in the trust chain prevent theft of their private keys, a valid ICE certificate chain serves as a sufficient non-forgeable and non-repudiable proof of trustworthiness for a device.

A device or app's metadata and runnable code will be signed by the manufacturer using the private key corresponding to the device model (or app equivalent) certificate. These signatures must be verified using the device model public key during the initial connection of an app or device in addition to verification of the certificate chain itself. Hence, authentication will fail for a device or app with code or metadata which does not match the certificate it presents.

The framework we propose will reduce the attack surface for ICE by making it more difficult to integrate malicious or otherwise unsafe devices and apps. Such apps or devices would ideally never be approved by the Certifying Authority, and it is computationally difficult (functionally impossible) to forge a CA signature, guaranteed in our system by cryptographic properties. While we believe this trust framework is necessary, we recognize this will not be sufficient to prevent dangerous or malicious behavior device or app.

This system will not significantly impede performance for devices or the middleware, as everything described here can be implemented using existing off-the-shelf cryptographic libraries using highly optimized code for well-known algorithms. This type of certificate chain verification occurs within web browsers every time a user visits a secure site using SSL/TLS. Likewise, verification of this certificate chain will not require any interaction the part of a clinician. The clinician would simply connect a device to the ICE system, or launch an app as they would otherwise. All trust verification will take place automatically within

[1] Note that "cloning" devices is out of scope for this paper, and can be managed through revocation lists.

the connection protocol for ICE compatible apps or devices.

In the remainder of this section, we describe each of the certificates in the trust chain. An overview of certificates and their relationships are shown in Figure 3.2. The detailed contents of the certificates is described below.

3.3.1 Root Certificate

PKI requires that a root of trust be established. We envision the root of trust as either a regulatory agency (e.g., FDA) or a third-party compliance/safety testing group. Unless otherwise stated throughout the rest of this paper, we will refer to the holder of the root certificate as the Certifying Authority. This certificate should be present in a list of trusted certificates as part of the ICE NC, similar to the way root certificates are distributed in web browsers. Otherwise, lacking a trust root, every certificate chain will fail to verify and connections will not be allowed.

3.3.2 Manufacturer Certificate

Device manufacturers possess an intermediate certificate issued by the Certifying Authority. The purpose of this certificate in the trust chain is to uniquely identify the manufacturer, and demonstrate that they have been recognized as a valid manufacturer by the Certifying Authority.

The certificate provides a non-repudiable claim by the manufacturer to have manufactured a particular device. To that end, the manufacturer credentials in the manufacturer certificate are used to sign device model certificates. Issuing machine-readable manufacturer certificates can augment the current medical manufacturer regulatory process, but neither replaces nor alters it.

3.3.3 Device Model Certificate

Each model of device has a distinct certificate called the device model certificate. This certificate uniquely identifies the model of device, is used as proof that it was created by a valid manufacturer, and proof that it meets any safety or regulatory criteria imposed by the Certifying Authority. A device model certificate is also associated with a metadata file for a device, which contains information including (but not limited to) the interoperable "data streams" the device produces and consumes, whether those streams are periodic or sporadic in nature, and access control information. Similar metadata is associated with runnable code for an app. The credentials contained in this certificate are used to sign individual device instance or app certificates, outlined below. This certificate is issued/signed by the device manufacturer, but also contains a signature from the Certifying Authority. We use a custom X.509 extension to implement this dual-signed certificate.

3.3.4 Device Instance Certificate

To uniquely identify each device, we use a device instance certificate. The primary identifying information for an individual device within ICE is it's globally unique identifier (GUID), which we store as the common name on this certificate. This certificate is an end-entity certificate, meaning that no other certificates may be issued using the private key for this certificate. Each individual medical device is an instance of a particular model of device, and to that extent, it is signed by the manufacturer using the corresponding device model certificate. Device instance certificates may be cached by the ICE NC to reduce the time required for future authenticated connections with that device.

3.3.5 Trust Chain Creation

A manufacturer who wishes to issue ICE compliant medical devices must first obtain a manufacturer certificate signed by the Certifying Authority. In order to do this, they send a certificate request to the Certifying Authority. The Certifying Authority, if they approve this manufacturer, sends back the signed manufacturer certificate. This process is simply an analog of the current regulatory requirements for medical device manufacturers, but produces machine-readable credentials. It neither replaces nor alters the current regulatory process.

Next we describe the creation of the device model and device instance certificates by walking through the ICE certification process for a newly developed medical device.

1. A manufacturer sends a prototype of a new medical device to the Certifying Authority as part of the process of collecting evidence of safety, effectiveness, and ICE compatibility. Along with this device, the manufacturer includes copies of the device's metadata and any associated runnable code (app) components and their metadata. Metadata files are signed by the manufacturer. Further, a signing request for the device model certificate is attached.

   The device model signing request is different from a standard certificate request, since the manufacturer will be the issuer of the device model certificate, while a Certifying Authority signature will be contained within a custom certificate extension on the device model certificate.[2]

2. The Certifying Authority analyzes and tests the device to ensure it meets compatibility and regulatory safety, security, and effectiveness standards. (The CA may conduct safety testing, or this may be done by a third party. For simplicity, we do not differentiate between the two cases.) When the device model is approved, the CA signs the device model signing request using its private key.

3. When an individual copy/instance of a device is created, the manufacturer creates a device instance certificate for that device. This certificate contains the globally unique

[2] The extension is a signature on the device model certificate. This will unambiguously indicate approval by the CA for this specific device model, since the device model is unique for each certificate.

identifier (GUID) for the corresponding device. It is then signed by the manufacturer using the private key corresponding to the device model certificate.

3.3.6 Authentication of Trust Chain

When a device connects to the MDCF, it presents its chain of certificates as evidence of trustworthiness and ICE and regulatory compliance. At any point in the verification process, if any single check fails, the device is prevented from connecting.[3] In addition to the steps outlined below, if the certificate falls outside of its valid date range, verification will fail. As an optimization, the Network Controller may cache certificate chains for devices which has previously connected. When reconnecting, a device can then send only its device instance certificate. However, if that certificate is not recognized by the NC, the device would have to undergo the full verification process.

The remainder of this section presents the steps for run-time verification of an ICE certificate chain for a device connecting to a Network Controller for the first time. This can be implemented in a standardized reference library and reused by any manufacturer.

1. The Network Controller (NC) first verifies that it already has a stored copy of the Certifying Authority (CA) root certificate included in this certificate chain. (For the moment, we assume a single regulatory authority, and thus a single root of trust.) If the root of the device's trust chain is an unknown CA, the connection to the device is terminated, and a system administrator is alerted. This device will be unable to connect to the ICE system unless its CA certificate is added to the NC's trusted certificate store.

2. Next, the NC validates the manufacturer certificate. The issuer field of the manufacturer certificate must match the subject field of the CA certificate. Furthermore, the signature on the manufacturer certificate must be verified using the public key from

[3] However, this verification may be bypassed in a "break-the-glass" scenario.[27]

the CA certificate.

3. Now, the NC verifies the device model certificate and checks code and metadata signatures. The issuer field of the device model certificate must match the subject field of the manufacturer certificate. Multiple signatures will be verified in this step. First, the signature on the device model certificate itself is checked using the public key from the manufacturer certificate, then the CA's signature on the device model certificate is verified using the CA's public key. Once the certificate is verified, the code and metadata signatures are verified using the public key from the device model certificate.[4]

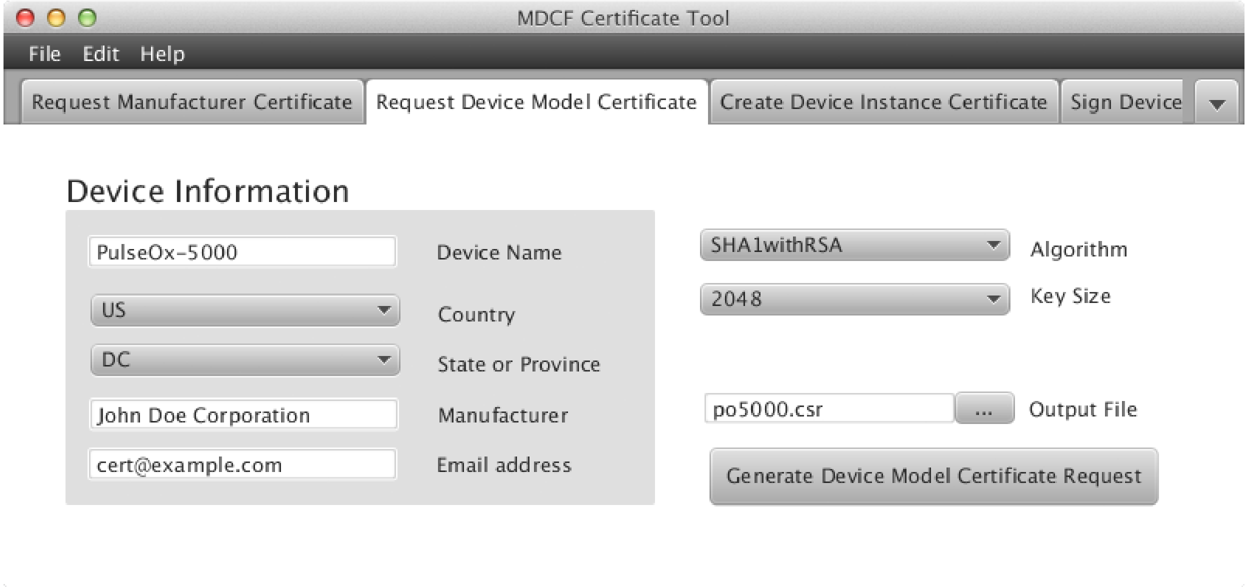
4. Finally, the NC determines whether the device instance certificate is trustworthy. In order for this to occur, the issuer field of the device instance certificate must match the subject field of the device model certificate. After this, the device model signature on the device instance certificate must be verified using the public key from the device model certificate. If the device instance certificate is accepted by the NC, this certificate chain is considered trustworthy, and is cached for future use.

Note that the entire verification process happens with **no user interaction**.

3.4 Certificate Authoring Tool

Although we expect certificates to be created in bulk, we nonetheless developed a stand-alone graphical certificate authoring tool to demonstrate the process of creating all the required certificates in the ICE trust chain (with the exception of the root of trust). This tool, pictured in Figure 3.3 is meant to be used by the Certifying Authority and by manufacturers. With it, a manufacturer can create certificate requests for their manufacturer certificate or a device model certificate, while the Certifying Authority can use it to sign these requests.

[4] This must be the last part in this step because the Network Controller would be unable to trust anything signed using the device model private key until after the device model certificate itself has been shown to be trustworthy.


MDCF Certificate Tool
File
Edit
Help
Request Manufacturer Certificate
Request Device Model Certificate
Create Device Instance Certificate
Sign Device
...
Generate Device Model Certificate Request

Device Information

PulseOx-5000 Device Name

US Country

DC State or Province

John Doe Corporation Manufacturer

cert@example.com Email address

SHA1withRSA Algorithm

2048 Key Size

po5000.csr Output File

Figure 3.3: *MDCF Certificate Authoring Tool – a GUI to streamline certificate creation for the ICE certificate chain. Pictured is the tab used for creating device model signing requests.*

Furthermore, manufacturers may use the tool to create the device model and device instance certificates themselves.

Chapter 4

Authentication

4.1 Motivation

Due to the threats (1.1) posed when an untrustworthy device is integrated into a MAP, an authentication system is needed to meet the security goals (1.3) of the MDCF (and MAPs in general). This system should be implemented in a way that allows medical device vendors to implement authentication providers that meet the needs of their devices. Furthermore, we intend for standardized, pre-made authentication modules to be made available by a centralized certifying authority or the ICE Alliance.[2] The MDCF should be extensible to ease integration of future technologies, and to make it easier to maximize compatibility by offering multiple security implementations. We use an approach similar to that of OpenSSL or Java security services by instantiating security providers by name instead of by a function call to a particular method.[7,8]

The modifications we describe in this section not only form the basis for adding a robust authentication system, but also serve to reduce the some of the burden on medical device developers by eliminating the necessity of creating an authentication system from scratch. The pre-defined modules that may be created using this framework can be used without

[0] The work presented in this chapter was originally published in a workshop paper titled "Retrofitting communication security into a public/subscribe middleware platform".[9]

modifying the MDCF.

We anticipate that this system will also allow us to create an authentication protocol that ties into the certificate framework described in chapter 3.

We begin this chapter with requirements for the authentication framework. Next we describe the design of this system. Finally, we discuss our evaluation of the framework.

4.1.1 Requirements

The MDCF device authentication framework serves as a layer of abstraction which frees developers from needing to modify the MDCF when implementing device authentication functionality.

We need this framework to enable automatic authentication of devices to occur without any intervention by clinicians. The authentication API should also be developer-friendly. Meaning it is mostly invisible to typical device developers — they should be able to use authentication without detailed knowledge of how the system works. Doing this requires some foresight into which specific components of the MDCF must be modified. The MDCF has a significant pre-existing code base (20K LOC), which we must hook into while also preserving compatibility with existing simulated medical devices. Thus, integration of the device authentication framework should be conducted while avoiding the need for major changes to the logical separation of components in the MDCF or fundamental architecture of the MDCF (particularly the connection state machine). We consider the "users" of this system to be developers of new MDCF-compatible devices. These users will make use of the framework in two different ways: using pre-made authentication providers on devices they develop, and creating new authentication providers. We have already implemented some providers and made them available to developers, but we also allow providers to implement their own authentication protocols. Lastly, we need to make sure that the performance overhead/resource utilization of the authentication framework is not significantly large enough as to interfere with normal operation of the MDCF.

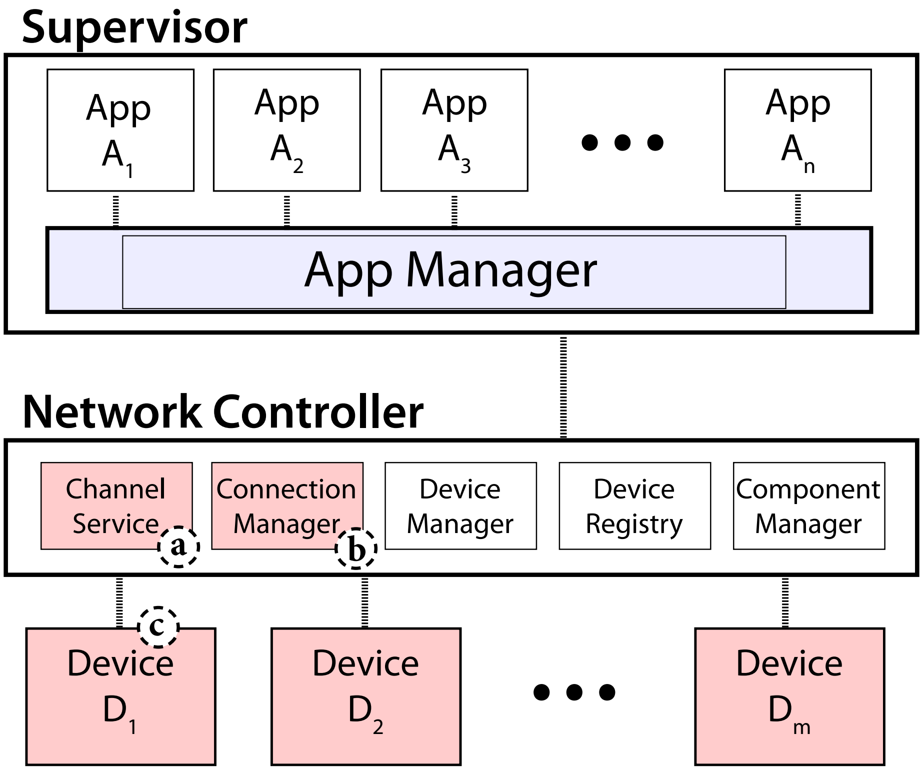


Figure 4.1: *MDCF components grouped by their logical ICE architecture role and showing primary hook locations (circles a,b,c).*

Requirements:

- the API should be expressive, powerful, and easy to use
- the framework should allow developers to implement arbitrary authentication protocols
- implementing authentication modules should not require alteration of the MDCF
- the only source of overhead should be the authentication modules themselves

4.1.2 Authentication Hooks

When we began this work, the MDCF was lacking any sort of security controls. Our first task was to modify the MDCF to place our authentication "hooks" in key locations in the code to allow us to later implement security modules. These modules are self-contained, reminiscent of SELinux.[28] In doing this, our goal was to guarantee that the security hooks are positioned in a way that allows developers to create "expressive" security modules. Our modifications to the MDCF code base are fairly small: the authentication framework (sans providers) is made up of a little over 1,000 LOC (MDCF as a whole was over 19K LOC originally). We evaluated the expressiveness of our hooks by initially developing a `NULL` authentication module (like IPSec NULL encryption).[29] This was followed by implementing other authentication modules including one using TLS authentication. Implementing these different authentication modules shows that the authentication framework is sufficiently flexible and expressive to implement nearly arbitrary authentication protocols, with effectively arbitrary rounds of messages being exchanged. All of this is done transparently to developers assuming they are using the pre-existing authentication providers in the MDCF.

4.1.3 Authentication Providers

Not only do we need to place security hooks for our authentication solution, we also need to create the components that comprise our actual security modules. These modules, called authentication providers, enclose the actual logic that makes up an authentication protocol. Authentication providers all implement the same interface so that we can easily drop in providers for different authentication protocols as needed. In this system, providers come in matched pairs. One provider in the pair is for the MDCF (server). The other provider is for a medical device (client). Providers are responsible for creating and connecting to dedicated authentication channels, as well as handling the sending and receiving of all messages in the protocol. Beyond serving as a container for the authentication protocol logic, the device-side authentication provider is also required to produce the data for the `authentication` message, which is the very first message sent from a device to the MDCF at the beginning of the device connection process. Contained within this message is information that identifies the protocol the device is attempting to use to authenticate with the MDCF. A device's metadata specifies which protocols that device supports. The provider that implements the chosen protocol is instantiated by name during device initialization. The MDCF also retrieves its corresponding authentication provider by name, based on the protocol specified by the device when it connects (assuming this protocol is supported by the MDCF).

4.1.4 Robustness and Resource Allocation

An important consideration when implementing authentication protocols is denial of service (DoS) attacks. Particularly, we need to pay attention to resource consumption, such as SYN flooding attacks on web servers.[30] We use a lazy resource allocation strategy to increase the robustness of the MDCF in the face of such attacks. To illustrate this, lets consider a possible first step during device connection: creating private channels for a device to communicate with the MDCF on (before authentication is complete). Because this occurs before authentication has completed, an attacker could create fake connection requests that cause resources to be allocated for private channels to devices that might never be allowed to fully communicate with the MDCF. In order to avoid this problem, we use only pre-allocated resources (pooled) until after a successful authentication has taken place. The allocation of resources during the connection process requires us to take special care in placing our authentication hooks. Although malicious devices might still eat up all of the resources in our pool, leaving it drained, we expect honest providers will eventually connect successfully.[16] Furthermore, this pool prevents the attack from interfering with existing device communications.

4.2 Security Design

4.2.1 Device Authentication Hooks

An AUTHENTICATING state originally existed in the communication state machine, however it was nothing more than a placeholder – the state, when entered, automatically transitioned to the AUTHENTICATED state. We placed our authentication hooks so that all authentication protocols are executed while in the AUTHENTICATING state, in order to make using our framework as seamless as possible for MDCF developers. Our hooks are such that a provider will execute on it's own thread while the state machine is in the AUTHENTICATING state, calling back to the connection state machine once authentication has finished which triggers a transition to AUTHENTICATED. In this way, we encapsulate and hide the multiple rounds of authentication message exchanges that occur within the AUTHENTICATING state. Consequently, entire authentication protocols can be implemented without needing to change the rest of the MDCF or device interfaces, as long as a communication channel from the MDCF Network Controller to the device has already been created.

Hooks are placed so that protocols are ran following the reception of the AUTH message by the Network Controller and before the AUTH_ACCEPTED message is transmitted All of these message are exchanged with either the DISCONNECTED or AUTHENTICATING states, making the connection state machine the primary location for the authentication hooks. The API we have created neither restricts the number of rounds nor the context of messages exchanged by authentication providers. Therefore, we claim that the framework meets the requirement of allowing developers to implement arbitrary authentication protocols. Due to the ability to implement arbitrary authentication protocols, developers are able to create providers that enable network controller (server) authentication as well as the standard device (client) authentication. At run-time, an authentication provider factory is created, as is a provider pool with at least one available provider when a device calls a provider during a connection attempt. Within the DISCONNECTED and AUTHENTICATING states, the provider is retrieved by name from the pool and executed. The AUTH message contains two pieces of information: a value specifying the authentication protocol requested by the device, and the device's public key (or certificates).

Not only did we need to create the hooks described above, we also had to add an additional message to the connection protocol, which is sent to the device by the Network Controller. This new message that we had to create is called the AUTH_PROTOCOL message. The purpose of this message is to transmit information to the device about the dedicated authentication channels that the Network Controller provider will use to carry out the authentication protocol. This message may optionally include additional information such as a public key for the Network Controller. Authentication providers exchange messages on the public atrium channels, `from_mgr_atrium` and `to_mgr_atrium`, before executing a protocol. These are the initial AUTH message and the AUTH_PROTOCOL message described above. The rest of the authentication protocol runs on the dedicated authentication channels sent in the AUTH_PROTOCOL message. We found it necessary to add these dedicated authentication channels to avoid unnecessarily broadcasting large quantities of messages on the atrium channels (all components on the MDCF receive these messages). Particularly when running simulations with large quantities of devices, the explosion in message traffic during authentication imposed prohibitively high performance overhead. Authentication providers execute their protocol when their respective `runAuthProtocol()` methods are called. (Note that this is not a security risk, as all messages between the MDCF and the device would be end-to-end encrypted and authenticated in a production system.) The connection state machines wait for these methods to return boolean values to indicate either a successful or failed authentication attempt. If the authentication fails then the device is disconnected, otherwise, the device resumes the connection process.

The location of the authentication hooks also allows us to ensure that authentication can not be bypassed (unless disabled totally by a system administrator). The only way

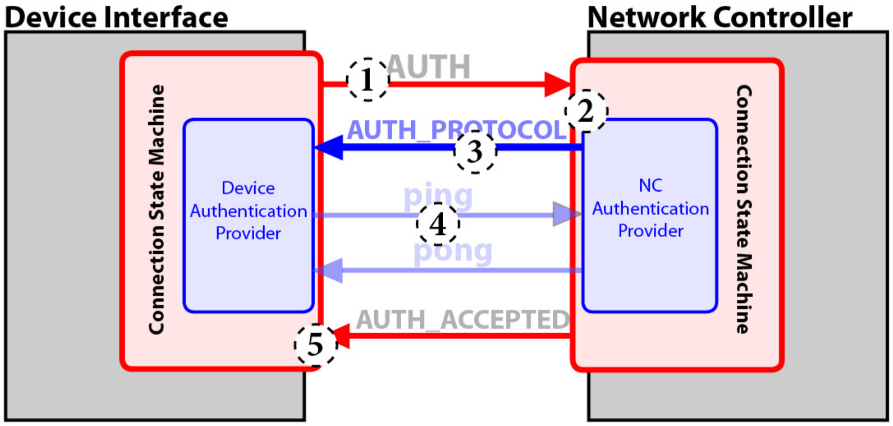


Figure 4.2: *Illustration of the authentication process using the* `NULL` *provider.*

that a device can connect to the MDCF is by progressing through the correct sequence of states in the connection state machine. The sequence of states can be seen in Figure 2.4. A device begins in the DISCONNECTED state. From there, it may only transition to the AUTHENTICATING state. A device in the AUTHENTICATING state must enter the DISCONNECTED state if authentication fails, or AUTHENTICATED if it succeeds. Each device must go through the AUTHENTICATING state in order to connect. The authentication hooks are placed such that an authentication provider must execute before a device can transition from AUTHENTICATING to AUTHENTICATED, therefore authentication may not be bypassed.

4.2.2 Authentication Providers

NULL

To check the overhead of our design and the expressiveness of the hooks, we implement a `NULL` authentication provider, which authenticates successfully if it receives a message "PONG" in response to its challenge "PING." The execution of the provider is mapped out in Figure 4.2 and described below.

1. The MDCF and device initialize. The MDCF initializes providers and populates its provider pool.

2. The device fetches any[1] supported authentication method from the device-side authentication provider object, then composes and sends an AUTH message to the Network Controller (NC) with the name of the authentication algorithm.

3. Upon reception of the AUTH message, the NC creates its own view of the connection state machine for this device. It then fetches the appropriate pre-initialized authentication provider from the provider pool, and passes the contents of the AUTH message to this provider.

4. The NC-side authentication provider obtains channels to be used for executing the authentication protocol with the device. It then sends an AUTH_PROTOCOL message to the device that specifies which channels should be used by the authentication protocol only (the device is assigned new channels after it successfully authenticates).

5. The protocol is executed, for the `NULL` authentication provider, which simply consists of an exchange of the strings "PING" and "PONG" between the device and NC.

6. Upon successful competition of the protocol, the NC creates new private channels for the device, sending the handles of those channels to the device in an AUTH_ACCEPTED message.

In addition to our `NULL` provider, we implemented three "non-trivial" providers for evaluation purposes: SSL/TLS, DSA, and DSA+DH. This exercise allowed us to confirm that we meet two of our stated requirements. We found that the API is sufficiently expressive, powerful, and easy to use. Also, as we explain later in this section, we found the only source of overhead comes from the authentication modules themselves.

[1] Multiple methods may be supported both by the device and the MDCF, but currently negotiation is not implemented.

SSL/TLS

Our TLS provider is based on Oracle's `java.net.ssl` implementation, running TLS 1.2 and using the `TLS_DHE_DSS_WITH_AES_128_CBC_SHA` cipher suite. This implementation can be trivially expanded to support mutual authentication with only a few additional lines of code (SSL/TLS provides this as standard functionality).

DSA and DSA+DH

The other two authentication providers (DSA and DSA+DH) use a simple challenge-response protocol in which a message from the device to the Network Controller includes a DSA signature from the device. Upon receiving this message, the Network Controller verifies the signature and then sends a signed response message to the device. Once the device verifies this response, the authentication protocol terminates – note that this is a mutual authentication protocol.

Provider Code Evaluation

Code length metrics for all providers are in Table 4.1. Compared to developing and implementing the framework, creating a provider is relatively simple, e.g. our SSL/TLS provider is only 207 lines of Java code. The compactness of these providers is beneficial – minimizing the amount of code and complexity within a security provider reduces the risk of introducing vulnerabilities stemming from implementation errors.

4.3 Evaluation

In order to confirm that we meet the requirement of overhead in the authentication system stemming only from the authentication modules themselves, we ran performance tests of our modified MDCF implementation on a server with dual hex-core Intel Xeon X5670 64-bit CPUs at roughly 2.93 GHz, with 12MB cache and 24GB system RAM, running Linux

Provider	**Implementation (LOC)**		**Increase over NULL**	
	MDCF	Device	MDCF	Device
NULL	128	72	1	1
DSA	151	120	1.18	1.67
DSA+DH	200	178	1.56	2.47
SSL/TLS	207	171	1.62	2.38

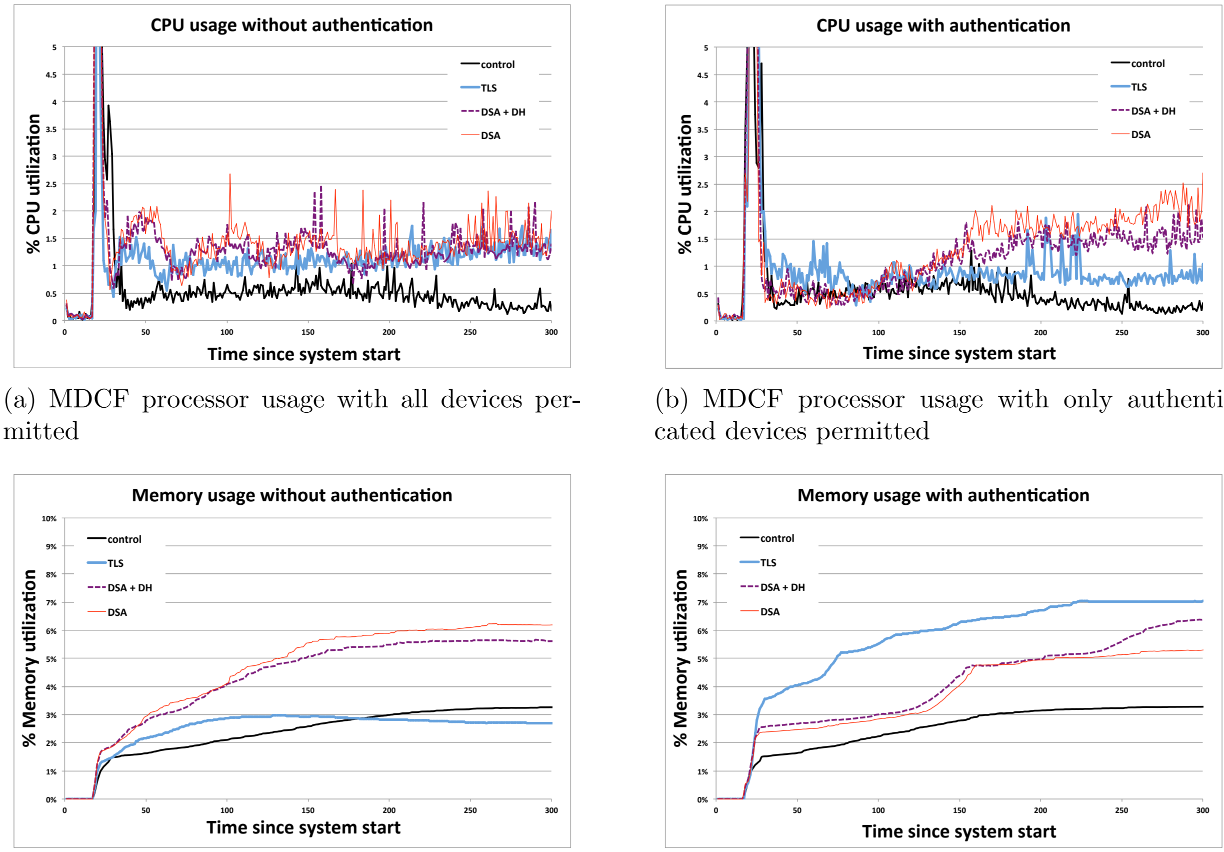
Table 4.1: *Authentication provider (device- and MDCF-side), complexity measured using lines of code (LOC), and complexity increase from the* NULL *provider.* NULL *is little more than the common infrastructure/scaffolding. The "Increase over* NULL*" column is therefore a more accurate representation of the code complexity increase of new authentication modules.*

3.8.13 and Sun's Java virtual machine version 1.7.0_21. The resulting performance is shown in Figure 4.3. Due to the limitations of Java and the current MDCF architecture on our test bed, we could only reliably test 340 or fewer concurrent devices. In an attempt to tax the resources of the MDCF and our authentication providers, the initial sharp spike in resource usage is due to all test devices attempting to connect simultaneously. Each device begins sending physiological data (SpO2 and pulse rate) following a successful join.

Figures 4.3(a) and 4.3(c) show the resource usage of MDCF using unauthenticated connections. The Y axis are constrained for readability. The highest observed CPU utilization within the start-up "spike" was 16% (DSA). Figures 4.3(b) and 4.3(d) show the resources used after including framework hooks only (control) and when using various authentication providers. The highest CPU utilization within the start-up spike was 19.75% (TLS), with DSA and DSA+DH reaching 17.4% and 16.7%, respectively. Each line represents an average of 11 instances of tests using identical configurations with 340 devices (device-side performance not shown for readability). The standard error is negligible (the difference between lines is statistically significant), and error bars have been omitted for clarity.

The control is a version of MDCF without any authentication code at all – the authentication code was not disabled, but rather removed entirely to avoid unexpected interactions.

The entire authentication framework consumes negligible resources – indistinguishable from control, satisfying our requirement. Authentication modules in Figures 4.3(a) and 4.3(c) show a modest but fixed resource cost. They are included in the running code, and a fixed



(a) MDCF processor usage with all devices permitted

(b) MDCF processor usage with only authenticated devices permitted

(c) MDCF memory usage with all devices permitted

(d) MDCF memory usage with only authenticated devices permitted

Figure 4.3: *MDCF resource usage with 340 virtual devices running on a different host.*

number are initialized to populate the provider pool, but are inactive – devices do not include authentication code and therefore never request to authenticate (backward compatibility mode). Running authentication modules impose an increase in resource usage dependent on the specific protocol being used (resource usage is protocol-dependent).

The network overhead in terms of latency and traffic volume is highly dependent on the individual protocol being used, and can be tuned (by selecting the appropriate protocol) depending on requirements. Authentication imposes a one-time latency increase due to the larger number of network round trips required at connection time, but the observable slowdown is negligible, and only occurs once – upon initial device connection. We found that only the TLS provider caused an increase in bandwidth usage, but to such a small extent as not to interfere with normal operation.

Chapter 5

Communication Security

5.1 Motivation

In order to prevent network-based attacks (1.2), the MDCF must have in place mechanisms for protecting data sent between devices/apps, and the MDCF server when necessary. We must ensure that patient information remains confidential. Furthermore, in order to protect against attacks using forged messages, the MDCF must be able to verify the authenticity and integrity of messages between devices, apps, and the MDCF server.

The purpose of the MDCF communication security framework is to serve as an abstraction layer which allows developers to implement different modules for communication security without having to modify the framework itself. Building such a framework therefore requires some foresight into which MDCF components need to be modified, and how to design the API to be developer-friendly and mostly transparent.

The MDCF communication security framework must hook in to the existing MDCF code base while maintaining backwards compatibility with existing devices. Integration of the communication security framework should not require significantly changing the fundamental design/architecture of the MDCF channel abstraction or otherwise disturb the overall logical separation of MDCF components – the incorporation of communication security should be mostly or completely transparent to developers working with (or modifying) the MDCF code or message transport layer (bus).

Our target "users" are developers working on new MDCF-compliant devices or apps. They will interact with this framework in two ways: using communication security providers on components they create, and/or by creating new communication security providers.

5.2 Requirements

In order to facilitate the creation of mechanisms for enforcing the security properties of confidentiality, integrity, and authenticity for communications in the MDCF, we have identified the following requirements for the framework components (API, hooks, etc).

- API is powerful, expressive, and easy to use
- Allow developer to perform arbitrary transformations on messages
- The only source of overhead should be the transformations themselves
- Using or implementing providers does not require altering the rest of the MDCF (outside of the framework)
- Communication security should be non-bypassable
- System should be invisible to clinicians
- Should be communication middleware-agnostic (not dependent on DDS or MIDAS)

5.3 Design

5.3.1 Hooks

To ensure that we can provide confidentiality, integrity, and authenticity of data in the MDCF and satisfy our implementation requirements, we must correctly position our com-

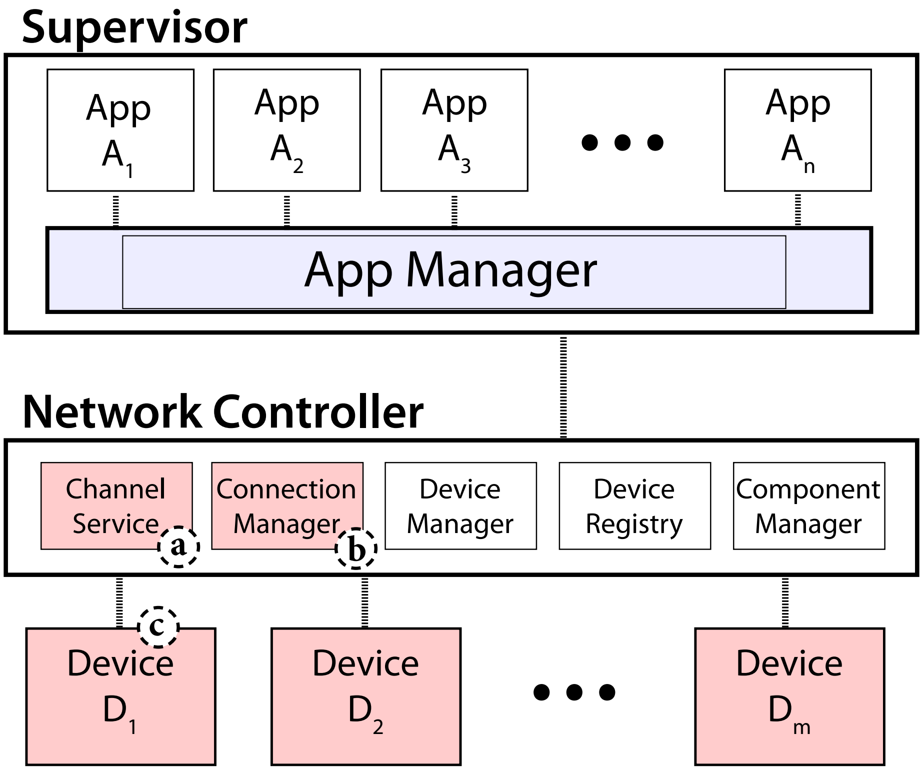


Figure 5.1: *MDCF components grouped by their logical ICE architecture role and showing primary hook locations (circles a,b,c).*

munication security hooks within the MDCF so that every message sent or received in the MDCF must pass through one of these hook locations. Specifically within the Channel Service, our hooks must fall within the Channel Abstraction. Although the Channel Service is pictured as being part of the Network Controller in (Figure 5.1), the Channel Abstraction sender/receiver API is available to all components of the MDCF, including the Supervisor, apps, devices, and the logger.

By placing our hooks there, we are able to create communication security providers that plug directly into the channel senders and receivers that all components within the MDCF must use to communicate. By tightly coupling our providers to senders and receivers, we ensure that the system will be non bypassable when in use.

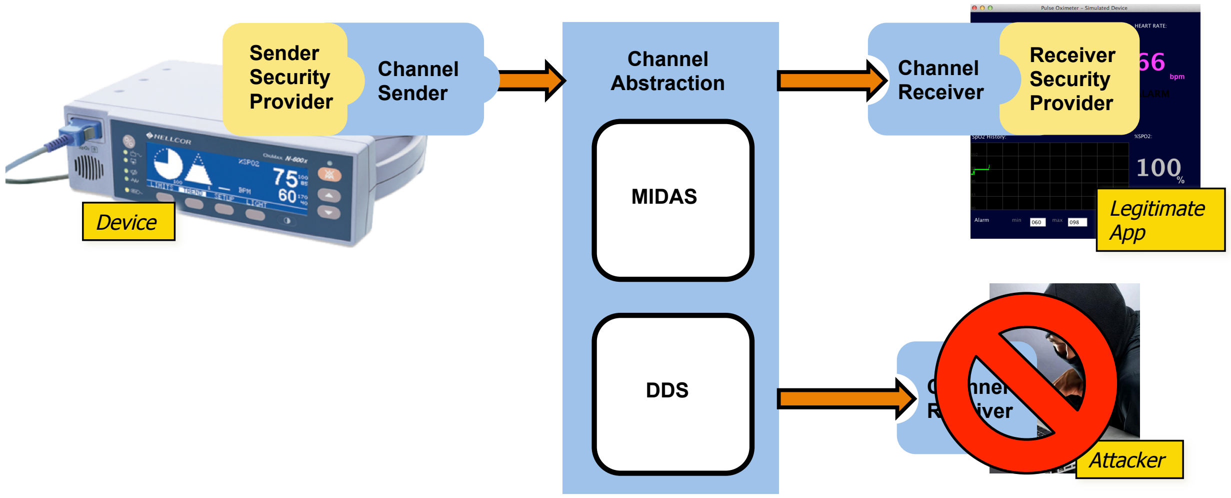


Figure 5.2: *MDCF channel security providers.*

5.3.2 Providers

These providers allow us to gain confidentiality through message encryption, and enforce and verify the integrity and authenticity of messages using digital signatures or message authentication codes.

Each channel security provider is able to perform arbitrary transformations on a message, so we can encrypt, decrypt, and authenticate messages. During the initialization of a message sender or receiver, it is bound to a channel security provider, ensuring that security-related transformations on messages may not be bypassed. Channel security providers come in pairs – one for sending a message and another for receiving a message. The application of these providers is on a channel-by-channel basis, making it possible to extend this feature to provide fine-grained control over how confidentiality, integrity, and authenticity are enforced for each channel.

Communication security is between a component and the network controller, a direct secure channel between a device and an app is not supported with this framework. There are several reasons for this. A device may publish data to multiple apps at the same time. Creating separate end-to-end secure channels between each device and the app would use additional device resources. Plain text copies of messages must be available to the Network

Controller for a logging system to function properly. This also allows us to implement the Role-Based Access Control system described in chapter (6), which must be able to sit between devices and apps so that access control can not be bypassed.

Building Secure Channels

Due to the need to properly generate and distribute keys for communication security providers, we envision that for devices and apps, the run-time selection of communication security providers will be driven by which authentication provider is in use. We believe that communication security and authentication providers may be developed together, leading to collections of providers analogous to TLS cipher suites. Currently, the communication security providers for a device or app communication session are currently selected according to which authentication provider is used.

The communication security framework we have built does not enforce a specific mechanism for distributing keys to the communication security providers. At this stage, we leave the logistics of key distribution and key agreement up to the developers of security providers. However, we have thought of a few possible ways to initialize communication security providers. One possibility is to derive all of the specific keys for each of the secure channels that the device or app will need a-priori during the connection-time authentication for those components. Alternatively, we can set up a secure management channel during authentication, which is kept open for the duration of the device or app connection to the MDCF. The Network Controller could then derive keys for other secure channels and send them across this channel. This is the approach we took with the non-trivial provider we implemented. Another possible approach is to obtain a single set of keys during authentication of devices or apps from authentication, and use those keys on demand to initialize the communication security providers between the Network Controller and those components. This second approach has the drawback of requiring additional key negotiations if we desire to assign a different key to each channel a device wishes to use.

5.4 Provider Implementation

To date, we have implemented two communication security providers. We first implemented a NULL communication security provider, which performs no operations on a message. Next, we implemented a more realistic provider which is based on Java's built in TLS implementation.

5.4.1 NULL Provider

The NULL provider served as the first step towards creating a usable interface and for ensuring that our hooks and framework correctly function before adding the additional complexity and overhead that come from performing cryptographic transformations of data.

5.4.2 TLS Provider

This TLS-based communication security provider uses a secure channel that was initially built during device authentication and was passed to the channel security providers from the TLS authentication provider. Specifically, it uses an instantiation of the java.net.ssl.sslEngine which was previously used for Authentication. The engine was restricted to using the

```
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
```

and was configured to exclusively allow TLS 1.2.

Chapter 6

Access Control

6.1 Introduction

In this chapter, we specify a Role-Based Access Control (RBAC)[1] system for the MDCF. This chapter contains two major sections. The first is a requirements specification modeled on the FAA's Requirements Engineering Management Handbook.[31] The second part of this chapter consists of remarks on the design for the RBAC system.

6.1.1 Background

Role Based Access Control

We believe that Role-Based Access Control extended with Break Glass functionality is the best option for controlling access to devices in apps in a MAP. As illustrated in Figure 6.1, RBAC relies on notions of users, roles and permissions. In this work, we look "Flat" RBAC,[1] one of the simpler models of RBAC, that is still considered to be a true RBAC system.

The following is a brief summary of how an RBAC system functions, based on our survey of the literature.[1, 10, 32, 33, 34] A role in RBAC is intended to correspond to the real-life roles

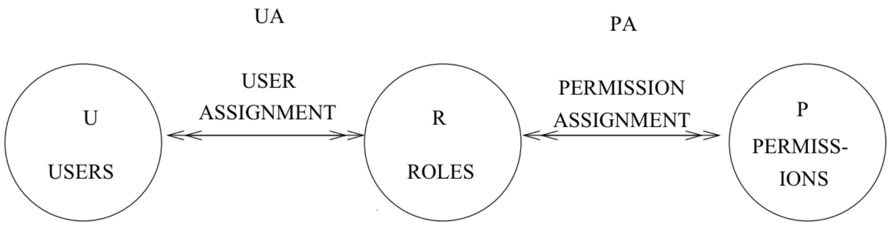


Figure 6.1: *Flat RBAC*[1]

that exist within an organization. Roles are meant have various permissions assigned to them. Permissions represent the ability to carry out some operation on an object. One of the benefits to RBAC is that permissions can be defined to be as fine or coarse grained as needed for a particular system. The permissions could be on the ability to read or write to some specific variable, or it could be a higher level operation such as the ability to connect to a channel on the MDCF. In some models of RBAC, permissions can specify that some specific action is taken or condition is met when the permission is used. This is called an *obligation*. On a basic level, the association of one or more permissions to a role is known as *permission assignment*. The systematic creation of roles and assignment of permissions to those roles, collectively known as *role engineering*, is recognized as a potentially difficult and time-consuming task and is outside the scope of this work. The association of a user to one or more roles, called *user assignment*, grants the user whatever permissions are associated with those roles. Note that in some RBAC systems, users must manually choose which roles they want to assume. This selection is made from the list of roles that user has already been assigned to, and is intended to help enforce the principle of least privilege.

Break Glass

Ideally, the policy defined for an RBAC system prevents users from acting maliciously or outside of their official duties without preventing them from accomplishing the tasks they are responsible for. However, the reality of the health care setting is that the environment can rapidly change – enforcing least privilege through explicitly defined access control polices can result in situations in which a clinician could be prevented from delivering potentially life-saving care to patients. In these situations, the need for a clinician to be able to administer care to a patient outweighs the risks associated with unauthorized access to medical devices or apps. Based on this, we conclude that the most suitable access control solution for the MDCF is an RBAC system extended with Break Glass features. The need for a Break Glass feature to bypass access controls in health IT systems has been recognized for years.[35,36] Combining RBAC and Break Glass, also sometimes called "Break The Glass" (BTG), has been examined before.[37,27] The simplest implementation is the creation of a specific user or role which holds what amounts to root permissions. Other implementations have been proposed that implement Break Glass separately from the rest of the access control system, and add granularity of Break Glass permissions[37] or context awareness.[38] A major theme of Break Glass systems is that an increase in freedom to access resources is coupled with a proportional increase in accountability for whatever actions are taken when the glass is broken. This enhanced accountability may include enhanced monitoring of the user's activities, and notifications of a supervisor or administrator.

Other Access Control Models

Pre-dating RBAC are Access Control Lists (ACL), Discretionary Access Control (DAC), and Mandatory Access Control (MAC). With DAC, users that have access to a given object, may delegate access to that object or its information.[32] The main problem with using DAC for an ICE-like system is that we do not want to allow access control policy decisions to be delegated to users. We prefer a centralized approach, in which an administrator defines all access control polices.

ACLs are commonly found in the file systems for operating systems. The idea behind ACLs is to attach a list to each file, with entries that indicate that a specific user or group can perform some action on the file.[32] The operating system only allows operations to be performed on this file if they have been white-listed in the ACL. It has been shown that ACL systems can be configured offer equivalent functionality to a “minimal” RBAC system.[39] However, distributing access control information as it is done with ACLs would likely be inefficient for ICE-like systems. We prefer RBAC to ACLs because it allows us to centralize access control information, which we believe will simplify system implementation and administration.

Mandatory Access Control is intended to control access in a centralized, non overridable manner.[32] A popular example in use today is SELinux, which is used to enforce the principle of least privilege on Linux processes.[28] All access control choices in a MAC system are defined by the system administrators, whereas in a DAC system, a user may be able to define the permissions of some object they own. Although MAC has properties that we desire, in that access control is centralized and mandatory. In its traditional form, we think that it is too inflexible for the needs of a medical environment. RBAC can provide the same benefits that MAC does, while remaining more flexible.[34]

Now, we will briefly discuss some newer access control models have been proposed since the popularization of RBAC. Attribute-Based Access Control has been proposed as a more flexible system than RBAC.[40] In ABAC, a user has a series of attributes associated with them. Furthermore, attributes may be based on the environment, or location, or an object itself. Access to an object is based on whether the user possesses the necessary attributes to perform an operation. The idea behind ABAC is to reduce the cost and complexity of using RBAC, by removing the need for role engineering and the assignment of specific permissions. It has been shown that it is possible to add attributes to RBAC, which may prove useful but is not addressed in this work.[40]

There is a variation of ABAC called Policy-Based Access Control (PBAC). PBAC is an attempt to extend the ABAC model to accommodate the enforcement of abstract policies (e.g. HIPAA) across large (enterprise) organizations.[33] One major difference is that the fundamental attributes required to access an object in PBAC are the same, regardless of whatever other local access controls are in place. Similarly, Risk-Adaptive Access

Control (RAdAC) evolved out of ABAC[41] so that context can be evaluated to assess risk when making access control systems. We believe that ABAC variations such as PBAC and RAdAC are not suitable for our needs owing to their complexity[33] and a lack of existing implementations.

6.2 Requirements

6.2.1 System Overview

The system being specified is the access control system for the Medical Device Coordination Framework (MDCF). Detailed information about the MDCF as a whole can be found in the Background section (2). The MDCF a high-assurance middleware system that serves as a Medical Application Platform. The MDCF also provides a platform for building medical apps that interact with devices in the form of receiving and viewing data issuing commands and changing settings.

In this section, we have built our requirements relying on a few basic assumptions about the MDCF. The devices are assumed to be connected to a single patient, and we assume that only one user will be actively controlling the system at any given time.

The MDCF consists of two major components — the Supervisor and the Network Controller (NC). The Supervisor is most easily viewed as the front-end. It hosts the user interfaces for the MDCF, including any MDCF apps. The Network Controller is essentially the server, routing, and back end components of the MDCF. Within the NC is the underlying message bus and any code related to connecting to and interfacing with devices. Devices and apps both have to connect to and authenticate with parts of the NC. The MDCF also supports data logging functionality, which works by copying messages off the message bus from within the NC, and then sending it off to the external logger.

The purpose of the access control system for the MDCF is to ensure that the MDCF, an MDCF app, or device being controlled through the MDCF only allows authorized individuals

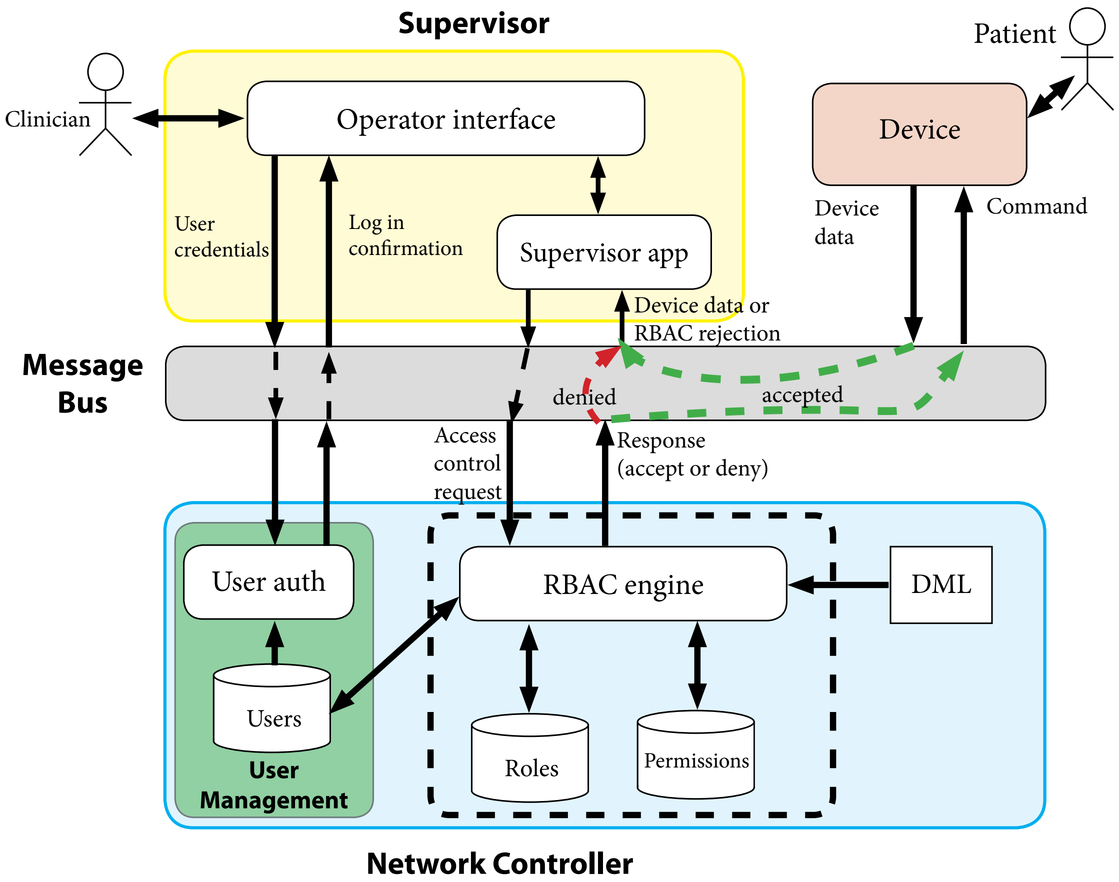


Figure 6.2: *Pictured is a context diagram for the RBAC system.*

to issue commands or view data. The access control system must also support a Break Glass feature that enables clinicians to bypass normal access control restrictions in emergency situations. This system will rely on a separate subsystem for managing user credentials and sessions. The access control system enables us to programmatically enforce clinical policy (and laws), the end result is that this system will be necessary for protecting patient safety, by helping ensure that only authorized clinical personnel are using the MDCF.

6.2.2 System Context

From Figure 6.2, we see that an RBAC system will interact with several other components of the MDCF. The RBAC system interacts directly with three entities that are part of the Network Controller: The user management system, DML metadata, and the Channel Abstraction/Message Bus. The RBAC system also indirectly interacts with the following entities outside of the Network Controller: Supervisor apps, Supervisor Console/Operator Interface, and ICE-Compatible devices

6.2.3 System Goals

Goal	Title
G1	Unauthorized users should not control devices
G2	Unauthorized users should not control apps
G3	Authorized users should be able to control devices
G4	Authorized users should be able to control apps
G5	Clinicians should be able to use a Break Glass feature
G6	System should operate with minimal user interaction
G7	System should operate with minimal performance overhead
G8	System should allow administrators to define custom access control policies

Table 6.1: *High-level goals(G) of the RBAC system*

6.2.4 Operational Concepts

We have created use and exception cases in order to describe how we conceive that various actors will interact with the MDCF and RBAC system. A Summary of the use cases is provided in Table 6.2, exception cases are in 6.3. Appendix B contains the use and exception cases listed in the tables. Furthermore, a list of the Actors in this system is located in Table 6.4. In the rest of this section, we examine one of the critical use cases of the system for illustrative purposes.

Use Case	Primary Actor	Description
B.1.1	Clinician	Normal Operation
B.1.2	Clinician	Break Glass Mode Operation
B.1.3	Clinician	User Log In
B.1.4	Clinician	User Log Out
B.1.5	Medical Device	Device Connects (First Time)
B.1.6	Clinician	Device Disconnects
B.1.7	Clinician	App Launch
B.1.8	Clinician	App Close
B.1.9	MDCF App	Issue Command to Device
B.1.10	Clinician	Enter Break Glass Mode
B.1.11	Administrator	Exit Break Glass Mode
B.1.12	Medical Device	(Break Glass) Device Connect (Unknown Device)
B.1.13	Clinician	(Break Glass) Device Disconnect
B.1.14	Clinician	(Break Glass) App Launch
B.1.15	Clinician	(Break Glass) App Close
B.1.16	MDCF App	(Break Glass) Issue Device Command
B.1.17	Administrator	Create Role
B.1.18	Administrator	Delete Role
B.1.19	Administrator	Add Role to User
B.1.20	Administrator	Remove Role from User
B.1.21	Administrator	Add Permission to Role
B.1.22	Administrator	Remove Permission from Role

Table 6.2: *Summary of use cases*

Issue Command to Device

Related System Goals: G3

Primary Actor: MDCF App

Precondition:

- MDCF is running
- Device is connected
- App is running and bound to Device
- Clinician is logged in

Exception Case	Primary Actor	Description
B.2.1	Clinician	User Log In (Failure)
B.2.2	Clinician	App Launch (Failure)
B.2.3	Clinician	App Close (Failure)
B.2.4	Clinician	Device Disconnects (Failure)
B.2.5	MDCF App	Issue Command To Device (Failure)
B.2.6	Administrator	Exit Break Glass Mode (Failure)

Table 6.3: *Summary of exception cases*

Actor Name	Primary Goals of the Actor
Clinician	Deliver care to patient by interacting with devices and apps
Administrator	Ensure that clinicians are able to deliver care to patients; Make sure hospital resources are used properly; Prevent anyone who is not a clinician at that facility from controlling devices or apps
Patient	Receive medical care administered by clinicians
Medical Device	Facilitate clinician's delivery of medical care by performing some medically relevant function (e.g. monitor patient physiological signals, dispense medicine)
MDCF App	Augment the capabilities of a medical device by displaying data published by that device, allow clinician to control and configure devices
RBAC Engine	Enforce access control policy for devices and apps in the MDCF

Table 6.4: *Actors in the RBAC system*

- Clinician is assigned to Role
- Role has Permission
- Permission is for desired command on Device
- The command has not been executed

Postcondition:

- MDCF is running
- Device is connected
- App is running and bound to Device
- Command has been executed

Main Success Scenario:

1. Clinician indicates that a command should be issued to Device within app
2. App requests permission to issue command to Device
3. RBAC Engine verifies that Clinician has Role and Role has Permission
4. RBAC Engine approves request
5. RBAC Engine forwards command to Device
6. Device executes command

The greatest need for access control comes from protecting patient safety by preventing untrusted/unprivileged individuals from controlling devices that can perform some actuation on a patient (e.g. administer morphine, respiratory ventilation). The use case we examine in this section presents a high level view of the interaction between an app and the RBAC engine that occurs when the app attempts to issue some sort of command to a device. This command could be anything from changing an alarm range value or to administering morphine through a PCA pump. Although the Clinician may be the one directly issuing a command in the app, this use case instead looks at how the app interacts with the RBAC Engine. Furthermore, commands might be either one-off sporadic commands directly originating from a user, or automated/scheduled commands from a workflow automation app. In this use case, we show that a user must be logged in, and that in order for their command to be executed, the role of that user must hold the permission matching the command they have attempted to run. In the case of an app autonomously sending commands, there are two possibilities for handling roles. One possibility is for apps act as users, holding their own roles. Another option is to control access to a channel that can be used to repeatedly issue such a command. Despite the importance of controlling access to devices, this is only one of many use cases involving RBAC.

6.2.5 External Entities

The following sections describe the external entities with which the MDCF RBAC system directly interacts along with any environmental assumptions made about each entity.

We assume that channels between the Supervisor and Network Controller, apps and Network Controller, and devices and Network Controller have mechanisms in place providing confidentiality, integrity, and authenticity of the data transmitted. These features are necessary to prevent eavesdropping, also in combination with message integrity/authenticity mechanisms, this prevents circumvention of access control mechanisms from forged messages and replay attacks. We need a standard way of storing and transmitting access control information for devices. Each device has a Device Modeling Language (DML) metadata file associated with it. This metadata file contains information about the features (i.e. data and commands) a device supports, making it the ideal location for access control information. These requirements assume the existence of access control information being embedded in the DML metadata for all devices. This information includes all possible permissions for a device, with default permission assignments to universally pre-defined roles. Furthermore, we assume that there is a component in the MDCF which parses access control data from DML and converts it into a format usable by the RBAC Engine.

We assume the existence of a user management system, including user databases, user authentication and tracking of user sessions. We need a database of users in the system, accessible by the RBAC Engine so that we can then do role assignment to those users and then drive our access control system based on role assignments and permission assignments to those roles. We assume that mechanisms are in place for authenticating users. Access control depends on a reliable user authentication system. Without it, there is no guarantee that the person associated with a user account is the intended person. We assume that the user management system keeps track of which user(s) are logged in at any given moment. We want to be sure of which user is using the MDCF, so that we can make correct access control decisions based on the roles and permissions associated with that user.

We assume that there is some mechanism that prevents individuals other than the clinician currently logged into to the MDCF from using the clinician's user account when that clinician is not actively using the MDCF Supervisor Console. We want to be sure that an unauthorized person cannot easily control apps and devices using the clinician's already logged in account (e.g. the clinician logs in, then steps out of the room briefly causing the Supervisor Console to automatically lock until that clinician returns or someone else logs in.) We assume that the Supervisor Console has some means of displaying feedback from the RBAC Engine to a user. We need to be able to inform users when an operation is prevented by the RBAC Engine, and when the system in in Break Glass Mode. We assume that there is a way, through the Supervisor Console to enter and exit Break Glass mode. Clinicians will primarily interact with the MDCF through the Supervisor Console, making it the convenient and logical location for the Break Glass controls. We assume the existence of an Administrative Console, either as a part of the Supervisor Console, or as a separate entity. The Administrative Console allows a system administrator to change MDCF configuration and security policy settings. An Administrative Console will allow administrators to define security policy without complicating the rest of the Clinician-facing Supervisor Console.

6.2.6 Requirements for the MDCF Access Control System

In this section, we define high level requirements for the Role-Based Access Control functionality. These requirements are duplicated in appendix B in a list format that may be easier for quickly finding individual requirements.

The first several requirements for our system are basic needs common to any RBAC system. Essentially, if we are going to create a Role-Based Access Control system, we need to be able to handle users, roles, and permissions. Specifically: *The RBAC system shall store and maintain roles; The RBAC system shall store and maintain permissions; The RBAC system shall store and maintain role assignments; The RBAC system shall store and maintain permission assignments.* These requirements ensure that we can maintain the basic information needed to make access control decisions using a Flat RBAC model.

The next collection of requirements we define are all related to the establishment and management of roles, permissions, user assignments, and permission assignments. Clinical environments where MAPs like the MDCF will be deployed may all have different organizational structures and operating procedures. Because of this, *administrators need to be able to create custom roles* to match the needs of their organization. In general, if we are going to support the creation and maintenance of custom access control policies, the system needs to provide some way to actually manage all of the access control artifacts (roles, permissions, users) and the relationships between those artifacts. Thus, *the RBAC system shall provide a means for managing roles, permissions, user assignments, and permission assignments.* So far, all of the requirements we have presented describe basic functionality essential to any RBAC system. However, the next requirement strays from the typical RBAC conventions — the MDCF must function as a plug-and-play system, so there needs to be a default set of roles that are universally present in any MDCF system or device. Without either common roles across all systems or some other universal way of grouping permissions, an unknown device may not be able to automatically work with a particular deployment of the MDCF. This is because there will be no way for it to comply with any custom access control policies enforced for that system. Therefore, *the RBAC system shall include default roles and default permission assignments for each device.* Also resulting from the plug-and-play nature of the MDCF, is our requirement that *the RBAC system shall automatically gather access control data from DML for devices and apps.* As we previously mentioned in the External Entities section, devices in the MDCF already have DML metadata files that contain information about the specific features that device offers and app DML specifies feature requirements. Thus, we should place our access control information in this DML file, in the form of permissions that correspond to device features which are assigned to the universal/default roles. For a device to automatically work with the MDCF, it will gather the permissions from DML for that device and either directly use the default roles, or perhaps rely on some set of heuristics for automatically assigning device permissions to custom roles.

The final collection of requirements specific to the RBAC part of our system are all related to specific actions that a user of a MAP might take which could potentially impact the safety or privacy of patients. The first of these requirements captures one of the main purposes of our system — we need to restrict the ability to control devices. A device may be responsible for some form of actuation on a patient (e.g. artificial respiration with a ventilator), so ensuring that the ability to send commands to devices is limited to authorized users is absolutely essential for patient safety. Hence, *The RBAC system shall prevent a command from being sent to a device unless the sender has a role with permission to send the command.* Because the primary form of interaction with devices and sensitive patient data is through apps, we also need to control the access to apps. Users should not be able to launch, use, or close an app unless they hold the privileges necessary for controlling the devices that app uses and viewing the data displayed by that app. Specifically, *the RBAC system shall prevent a user from launching, using, or closing an app unless the user has a role with permission to launch the app.* Apps may potentially be responsible for ongoing or repetitive tasks that are medically relevant, thus closing an app can be a critical operation that should be restricted. For exactly the same reason, *the RBAC system shall prevent a user from disconnecting a device unless the user has a role with permission to disconnect the device,* since devices may be associated to apps and may be executing some sort of medically relevant task that relies on its connection to the MDCF.

6.2.7 Requirements for Break Glass System

The Break Glass function bypasses the normal access restrictions imposed by the Role-Based Access Control function, allowing clinicians to quickly provide care in an emergency situation that deviates from what is typical in the environment the MDCF is deployed in. The main difference here from other Break Glass systems comes down to this being used to control access to medical devices on a publish-subscribe middleware system.

The first requirement is fairly obvious: There needs to be a way to actually "break the glass" so that we can enter Break Glass mode. In other words, *the Break Glass system shall allow users to activate Break Glass mode.* Similarly, we need some way to exit Break Glass mode. As we discussed in the last section, one of the most important functions of the RBAC system is to restrict control of devices. This restriction of device control may be problematic if, for instance, a patient is having an emergency and the only clinician near them does not have the permissions needed to control some device needed to address the problem. Because of this, *the Break Glass system shall allow full, unrestricted control of devices while in Break Glass mode.* For the same reason, in order to deal with an emergency/exceptional event, *clinicians should be able to launch, use, and close apps as needed to deliver care to the patient while the MDCF is is in Break Glass mode*, regardless of their normal roles.

Our last requirement is a result of the architecture of the MDCF — components must communicate across channels and functionality is somewhat decoupled. The Break Glass system needs to notify other components of the MDCF that a Break Glass scenario is occurring so that the other components may execute whatever special Break Glass functions they are responsible for. Specifically, *the Break Glass system shall broadcast a notification when Break Glass mode is entered or exited.* A motivating example for this broadcast is that the logger may need to behave differently during a Break Glass scenario by recording more information so that users are held accountable for their actions.

6.3 Design

Design, in this section, refers to a high level description of the features provided the RBAC system, the location of system components within the conceptual architecture of the MDCF, and the relationships between components which implement those features. This section explores the design-related concerns of the access control system which came up during the creation of the requirements section. The intention of this is assist future efforts in building an actual access control system in the MDCF or other MAPs. Our proposed design is split into different levels, each level builds upon the lower levels, and unless otherwise stated, whatever applies to Level 1 also applies to Level 2 and so on. This proposed design would would be implemented in stages corresponding to these levels.

6.3.1 Level 1: Minimal Implementation

We believe that the initial implementation of the access control system should be kept as simple as possible. Thus, we suggest that a Flat RBAC (6.1) model is used at least initially. This initial implementation will only support a set of pre-defined roles (e.g. Administrator, Clinician, Surgery, Anesthesiology).

Break Glass can be initially implemented as an additional default role, which holds all permissions. The assignment to this role would be temporary, and should be removed from the user when Break Glass mode is executed. When Break Glass mode is activated or deactivated, an announcement should be broadcast to other components so that they can execute Break Glass logic (i.e. enhanced logging, user interface indicator, notify clinic supervisor). DML for devices should contain an annotation which enumerates all the permissions for the device. Permissions may initially be defined in terms of the ability to either read/write/execute some DML capability, or in the ability to subscribe to the exchanges necessary to access device capabilities. Similarly, just as the DML for apps specifies capability requirements for that app, so will the access control requirements for the app be derived. If the app requires some set of capabilities to operate, then it also requires that a user launching, using, or closing that app to have permissions to use those capabilities. If the user lacks permissions to use all features of the app, they are not allowed to access it at all. This should not be a problem, since only one user can use the system at a time at this level. If the user logs out, all of the apps they launched will be closed. In this system there are no databases aside from perhaps a user database. The default roles can be hard coded into the system. Permissions and role assignments are pulled from the DML for devices and apps when they are launched/connected. These permissions are stored in memory while the device is connected, but are in not permanently cached/stored (there are no databases).

For now, we will assume that the Supervisor and Network Controller are both running on the same hardware. Since there is only a single user, and the user management system controls the initial access to the Clinician Console, we can simply query the user management system to see who the current user is when an access control decisions must be made. A more distributed version of the MDCF, might rely on some sort of proof of identity such as a token issued by the user management system that is unique to a given user for a given session (and which expires after a set amount of time). However, design of that system is outside of the scope of this work.

At this level, because only a single user is ever able to use the system at a time — access control can be limited to restrictions on launching devices. The access control system here only does two things: control the launching of apps and support rudimentary Break Glass. No administrative support features are necessary at this initial level because the access control policies can not be customized.

6.3.2 Level 2: Customized Local Security Policy

The next step in implementing the MDCF access control system is to support the creation of customized security policy. Specifically, this means that an administrator is able to create roles and assign permissions to those roles.

This level requires the existence of Administrative tools for managing custom local RBAC policy. Furthermore, this requires storage of roles and permissions by the RBAC system. Storage of roles and permissions points to the addition of a database to the RBAC system. With this storage in place, at this level, we would also start to cache the permissions of devices during their first connection to the MDCF, removing the need to gather this information from devices during subsequent connections. The intention here is to speed up the connection process for "known" devices, similar to what we propose in the certification framework (chapter 3).

The default roles (including Break Glass) from Level 1, would also be stored in the databases, replacing the original hard-coded implementation. Access control decisions will be made entirely based on data retrieved from the user management system and the role and permission storage. Because of this, local policy either has to be defined by directly manipulating the policy databases, or through some intermediate format. One possibility is to adopt the eXtensible Access Control Markup Language (XACML)[42] or some subset of it for defining access control in the MDCF. XACML is a standardized markup language for declaring access control policy, and has been used by other systems combining RBAC and Break Glass features.[27][37] If XACML or some other intermediate representation of access control policy information is used, the Administrative Console of the MDCF will need to facilitate this, which may prove to be a significant undertaking on its own.

Plug and play with custom policies

A customized security policy in the MDCF raises the issue: how do we support plug and play functionality for unknown devices and apps while enforcing a customized security policy? An administrator could try to define policy up-front for some set of generic DML schemas for common classes of devices and apps. However, if new types of components are connected, or if some component offers additional functionality, there needs to be some way to deal with that. We either need to support the use of components for which users only have partial permissions, or come up with some automatic way of performing permission assignment to custom roles.

6.3.3 Level 3: Multiple Users

At this level we introduce multiple simultaneous users. The main difference here is that a user can launch an app and leave it running while another user accesses the system too. This, of course, relies on a user management system that supports this. For instance, one user logs in and launches an app and then steps away/locks the console. After this happens, a different user logs in, and accesses the app(s) launched by the previous user.

An app is only launched if the user launching it has the permissions the app requires. For another user to access that app, they must also have those permissions. This will require the creation of a message format for access control requests. The message should contain:

- User ID
- User session token (place holder)
- Subject of the request
- Unique ID for operation requested

Although at this level, it may only be used for gaining access to apps launched by other users, this access control message may be more important if finer-grained access control features are to be implemented later on. A system like we describe here has very coarse-grained permissions, a more complex system may allow users to have partial access to device or app features. This sort of system may require access control decisions to be made for each command when the app or device is only partially accessible to a user.

6.3.4 Level 4: More Granularity

The system described in the previous levels provides very coarse grained access control. Accommodating partial access to apps and devices would give us more granularity. An even finer grained system could be created by extending our RBAC system with attributes.[40] For example, there could be attributes which capture which part of a care facility the clinician is assigned to, or that the clinician is scheduled to be working that day. These attributes are determined by querying a scheduling system at the care facility. The access control requirements for a device or app could then be extended to specify that only the clinicians assigned to that patient, that are currently scheduled to be working are able to use devices or apps connected to that patient.

The system might also be expanded at this level to support a more complex version of RBAC, by explicitly handling user sessions and obligations. The system described in the previous levels announces that glass has broken, and then we assume other components respond appropriately. This interaction could be more explicitly represented as an obligation attached to Break Glass permissions.

The Break Glass system could also be made more granular. Instead of granting unrestricted access to everything right away, we could implement a system that only grants access to specific features requested by the glass-breaker. Two different approaches to this can be seen in the BTG-RBAC Model,[37] and Rumpole.[38]

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Although the use of Medical Application Platforms may result in a net improvement to patient safety, they also carry risks to patient safety and privacy. We have described a collection of security features for MAPs in order to address some of these risks. The work on these security features is meant to serve as the starting point for a comprehensive security architecture for MAPs.

We began by presenting a system for ensuring the trustworthiness of medical devices connecting to the MDCF. In order to prove that a device is trustworthy, we use a chain of X.509 certificates that serve as non-forgeable proof of regulatory approval, safety testing, compliance testing, and device identity. Next, we described the design and implementation of a flexible, modular device authentication system into the Medical Device Coordination Framework. We evaluated our authentication system by examining the complexity and performance of proof-of-concept device authentication providers. We then discussed the creation of a communication security framework in the MDCF. This system facilitates the implementation and deployment of communication security providers that provide data confidentiality, integrity, and authenticity. The final topic we addressed in this work was access control. Specifically, we defined requirements and a high level design for a system combining Role-Based Access Control (RBAC) with Break Glass features.

7.2 Future Work

A great deal of work remains to be done in order for this security architecture to be fully realized. Much of this work requires additional implementation and/or evaluation work to be carried out. The certificate authoring tool we described at the end of chapter 3 needs to be completed. Server-side (Network Controller) verification of certificate chains in the MDCF also remains to be implemented. This could be implemented using a specially built pair of authentication providers.

The communication security system needs the implementation of practical communication security providers that may be used when TLS authentication is not used. Furthermore, we believe that the communication security framework should include additional features to streamline the handling of cryptographic keys. This would likely involve storing and retrieving keys from the MDCF device registry. We did not look into how authentication, communication security, and access control systems will interact with the real-time properties of the MDCF. This interaction between real-time guarantees and security may warrant additional evaluation and implementation efforts.

Our contribution on the topic of access control was limited to studying the background, drafting initial requirements, and describing a preliminary design for the system. A more complete design for the access control system needs to be created. This design would ideally address the issue of mapping new device or app permissions to the roles that exist in a custom access control policy. It should also define the means by which customized access control policies are defined (e.g. XACML).

After the creation of a more detailed design, the system will need to be implemented. The design and implementation efforts for the access control system will help verify the work presented here and may also lead to a revision or extension of the requirements we have presented. Once the access control system is implemented, a thorough evaluation will be necessary.

Before any meaningful implementation of access control can be carried out, some additional features need to be added to the MDCF. First of all, a system for managing user credentials, logins, and sessions needs to be implemented. The access control system directly depends on such a system. Next, there needs to be support for access control functionality within the user interface components (Supervisor/Clinician Console). These UI additions may be implemented in steps corresponding to the underlying access control system features they support.

Furthermore, work to ensure that logging is handled in a privacy and accountability preserving matter needs to be conducted. Future work may also address how to leverage the security features of the MDCF and the logging system to create strong mechanisms for enforcing accountability (i.e. non-repudiation).

Bibliography

[1] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The nist model for role-based access control: Towards a unified standard. In *ACM Workshop on Role-based access control*, 2000.

[2] John Hatcliff, Andrew King, Insup Lee, Alasdair MacDonald, Anura Fernando, Michael Robkin, Eugene Y. Vasserman, Sandy Weininger, and Julian M. Goldman. Rationale and architecture principles for medical application platforms. In *International Conference on Cyber-Physical Systems (ICCPS)*, 2012.

[3] John Hatcliff, Eugene Vasserman, Sandy Weininger, and Julian Goldman. An overview of regulatory and trust issues for the integrated clinical environment. In *Proceedings of the Joint Workshop On High Confidence Medical Devices, Software, and Systems & Medical Device Plug-and-Play Interoperability (HCMDSS/MD PnP)*, 2011.

[4] Andrew King, Sam Procter, Daniel Andresen, John Hatcliff, Steve Warren, William Spees, Raoul Jetley, Paul Jones, and Sandy Weininger. An open test bed for medical device integration and coordination. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 141–151, 2009.

[5] David Arney, Sandy Weininger, Susan F. Whitehead, and Julian M. Goldman. Supporting medical device adverse event analysis in an interoperable clinical environment: Design of a data logging and playback system. In *International Conference on Biomedical Ontology (ICBO)*, July 2011.

[6] E.Y. Vassserman and J. Hatcliff. Foundational security principles for medical applica-

tion platforms. In *Proceedings of the International Workshop on Information Security Applications*, August 2013.

[7] Li Gong and Gary Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2nd edition, 2003. ISBN 0201787911.

[8] OpenSSL. OpenSSL: Documents, ssl(3). https://www.openssl.org/docs/ssl/ssl.html, January 2012.

[9] C. Salazar and E.Y. Vassserman. Retrofitting communication security into a publish/subscribe middleware platform. In *International Workshop on Software Engineering in Health Care (SEHC)*, July 2014.

[10] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. In *IEEE Computer*, volume 29, 1996.

[11] ICEMDPnP. Medical Device Plug-and-Play (MD PnP) Integrated Clinical Environment (ICE) website. http://mdpnp.org/ICE.html, 2014.

[12] ASTM Committee F-29, Anaesthetic and Respiratory Equipment, Subcommittee 21, Devices in the integrated clinical environment. Medical devices and medical systems — essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE), 2009.

[13] Andrew King, Dave Arney, Insup Lee, Oleg Sokolsky, John Hatcliff, and Sam Procter. Prototyping closed loop physiologic control with the medical device coordination framework. In *Proceedings of the ICSE Workshop on Software Engineering in Health Care (SEHC)*, pages 1–11, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-973-2. doi: 10.1145/1809085.1809086.

[14] MDCF. Medical Device Coordination Framework (MDCF) project. http://mdcf.santos.cis.ksu.edu.

[15] B. Snyder, D. Bosanac, and R. Davies. *ActiveMQ in Action*. Manning Pubs Co Series. Manning Publications, 2011. ISBN 9781933988948. URL http://books.google.com/books?id=_jjCPwAACAAJ.

[16] J.K. Millen. A resource allocation model for denial of service. In *IEEE Symposium on Security and Privacy*, 1992.

[17] K.K. Venkatasbramanian, S.K.S. Gupta, R.P. Jetley, and P.L. Jones. Interoperable medical devices communication security issues. In *IEEE Pulse*, September 2010.

[18] K.K. Venkatasbramanian. The chronicles of interoperability: Failures, safety, and security. In *AAMI Horizons*.

[19] K.K. Venkatasbramanian, E.Y. Vassserman, O. Sokolsky, and I. Lee. Security and interoperable-medical-device systems, part 1. In *IEEE Security and Privacy Magazine*, November 2012.

[20] E.Y. Vassserman, K.K. Venkatasbramanian, O. Sokolsky, and I. Lee. Security and interoperable-medical-device systems, part 2: Failures, consequences and classification. In *IEEE Security and Privacy Magazine*, September 2012.

[21] D.F. Kune, K.K. Venkatasbramanian, and E.Y. Vasserman. Toward a safe integrated clinical environment: A communication security perspective. In *Workshop on Medical Communication Systems (MedCOMM)*.

[22] C.R. Taylor, K.K. Venkatasbramanian, and C.A. Shue. Understanding the security of interoperable medical devices using attack graphs. In *Conference on High Confidence Networked Systems (HiCoNS)*.

[23] Food and Drug Administration. Unique device identification (UDI). http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/UniqueDeviceIdentification/, 2014.

[24] Internet Engineering Task Force. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. http://tools.ietf.org/html/rfc5280, 2008.

[25] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011. ISBN 978-1-4503-1000-0. doi: 10.1145/2046614.2046618. URL http://doi.acm.org/10.1145/2046614.2046618.

[26] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, Second Edition*. Wiley, 2008.

[27] Achim D. Brucker and Helmut Petritsch. Extending access control models with break-glass. In *Proceedings of the ACM symposium on Access control models and technologies*, 2009. ISBN 978-1-60558-537-6. doi: 10.1145/1542207.1542239. URL http://doi.acm.org/10.1145/1542207.1542239.

[28] Bill McCarty. *SELinux: NSA's open source security enhanced Linux*. O'Reilly, 2005.

[29] R. Glenn and S. Kent. The NULL encryption algorithm and its use with IPsec, 1998.

[30] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on TCP. In *IEEE Symposium on Security and Privacy*, 1997.

[31] Federal Aviation Administration. *Requirements Engineering Management Handbook*. 2009.

[32] Ravi S. Sandhu and Pierangela Samarati. Access control: Principles and practice. In *IEEE Communications Magazine*, 1994.

[33] NIST. A survey of access control models. In *Privilege Management Workshop*, 2009.

[34] Qamar Munawer Sylvia Osborn, Ravi Sandhu. Configuring role-based access control to enforce mandatory and discretionary access control policies. In *ACM Transactions on Information and System Security*, volume 3, 200.

[35] How to break access control in a controlled manner. A. ferreira and r. cruz-correia and l. antunes and p. farinha and e. oliveira-palhares and d.w. chadwick and a. costa-pereira. In *IEEE Symposium on Computer-Based Medical Systems (CBMS)*, 2006.

[36] HIPAASECURITY:URL. Break Glass Procedure: Granting Emergency Access to Critical ePHI Systems. http://hipaa.yale.edu/security/break-glass-procedure-granting-emergency-access-critical-ephi-systems, 2014.

[37] Ana Ferreira, David Chadwick, Pedro Farinha, Ricard Correia, Gansen Zao, Rui Chilro, and Luis Antunes. How to securely break into rbac: the btg-rbac model. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[38] Srdjan Marinovic, Robert Craven, Jiefei Ma, and Naranker Dulay. Rumpole: A flexible break-glass access control model. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2011.

[39] John Barkley. Comparing simple role based access control models and access control lists. In *ACM workshop on Role-based access control*, 1997.

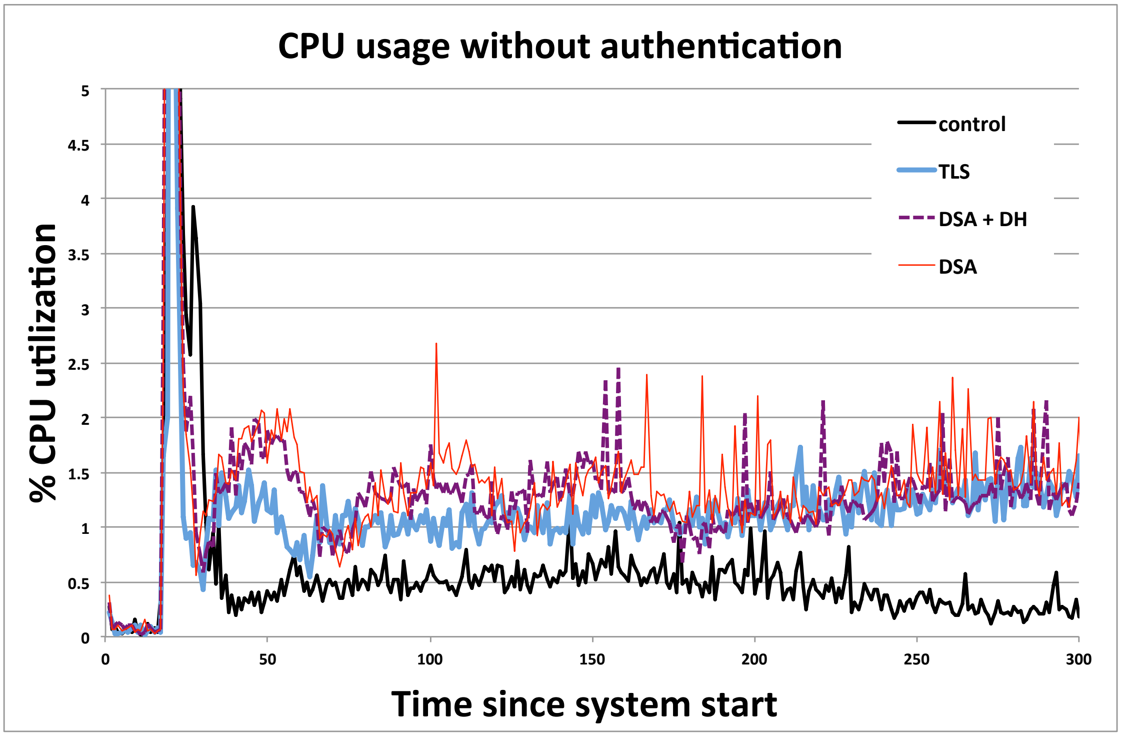
[40] D. Richard Kuhn, Edward J. Coyne, and Timothy R. Weil. Adding attributes to role-based access control. In *IEEE Computer*, volume 43, 2010.

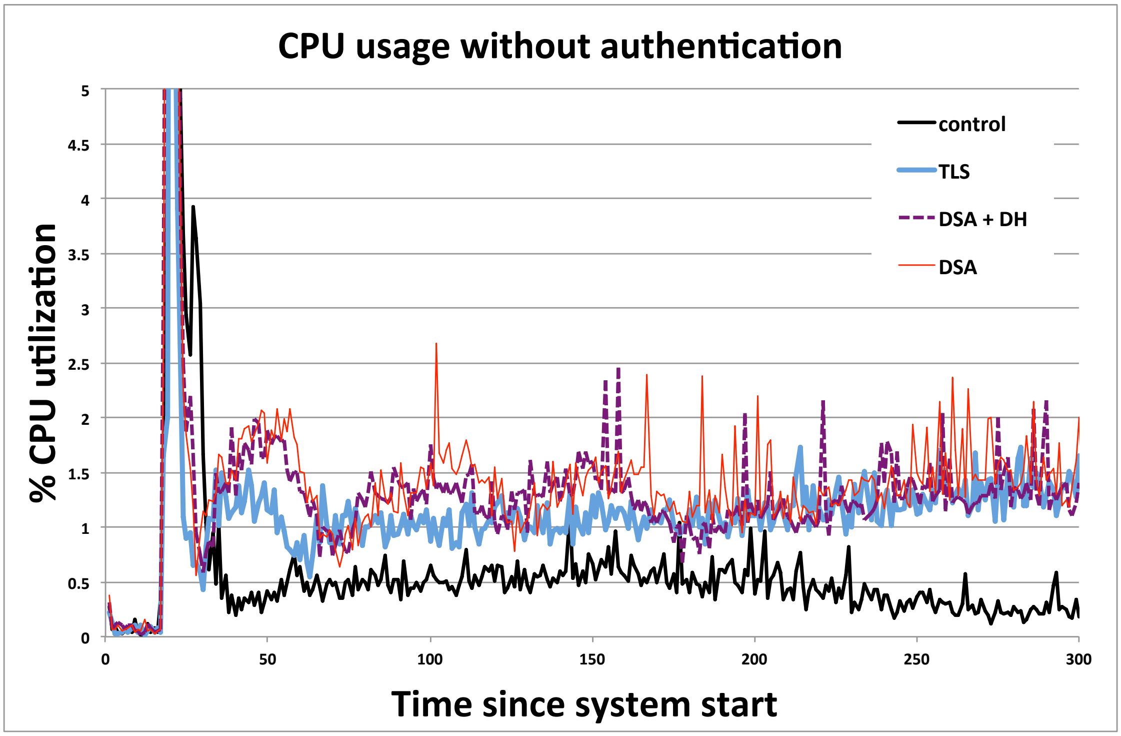
[41] Michael H. Davis Alan H. Karp, Harry Haury. From abac to zbac: The evolution of access control models. In *HP Laboratories Tech Report*, 2009.

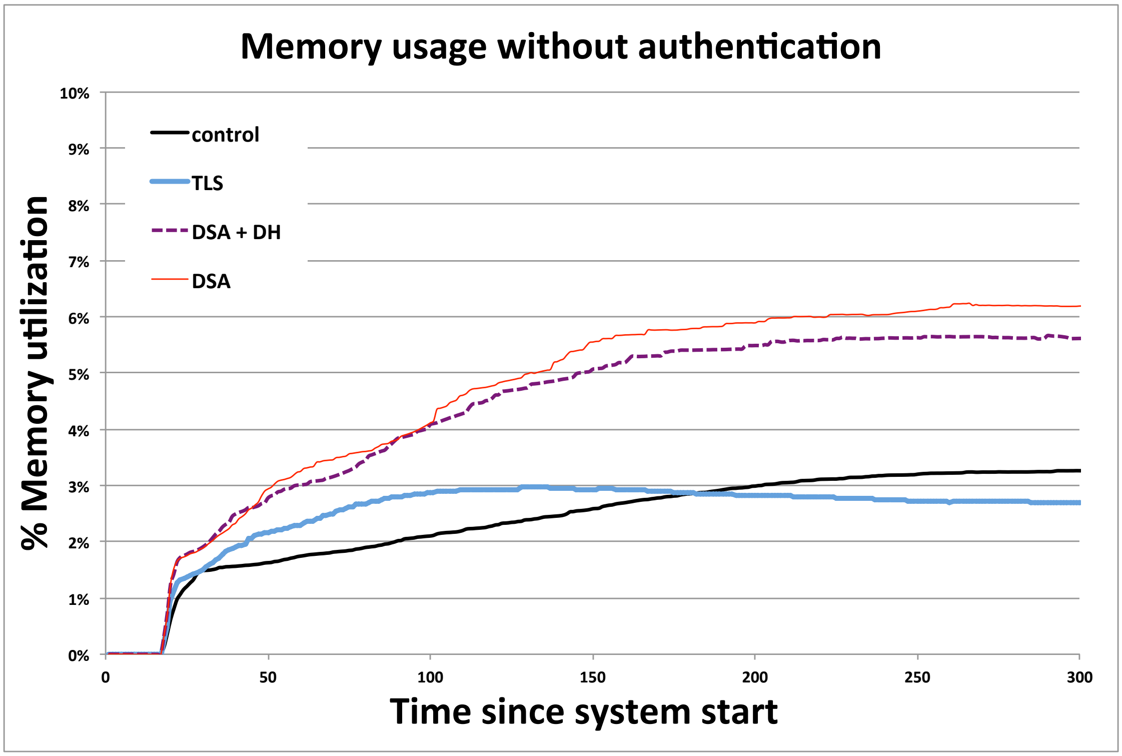
[42] OASIS. OASIS eXtensible Access Control Markup Language (XACML) TC. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml, 2014.

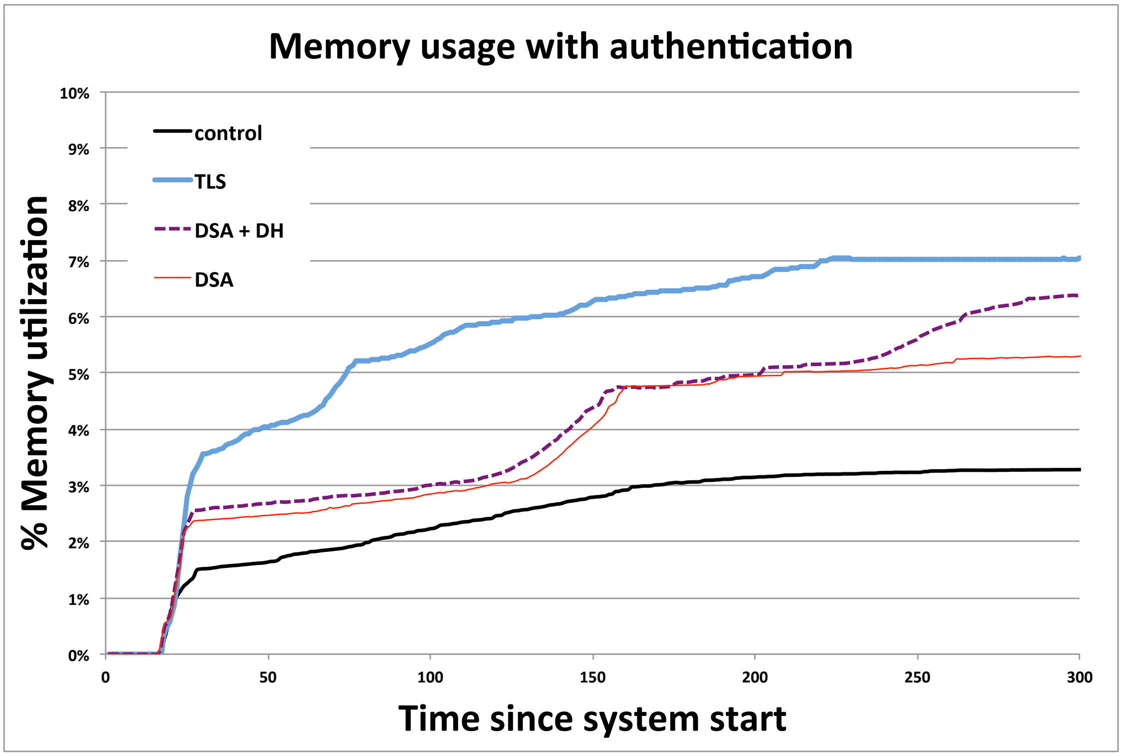
Appendix A

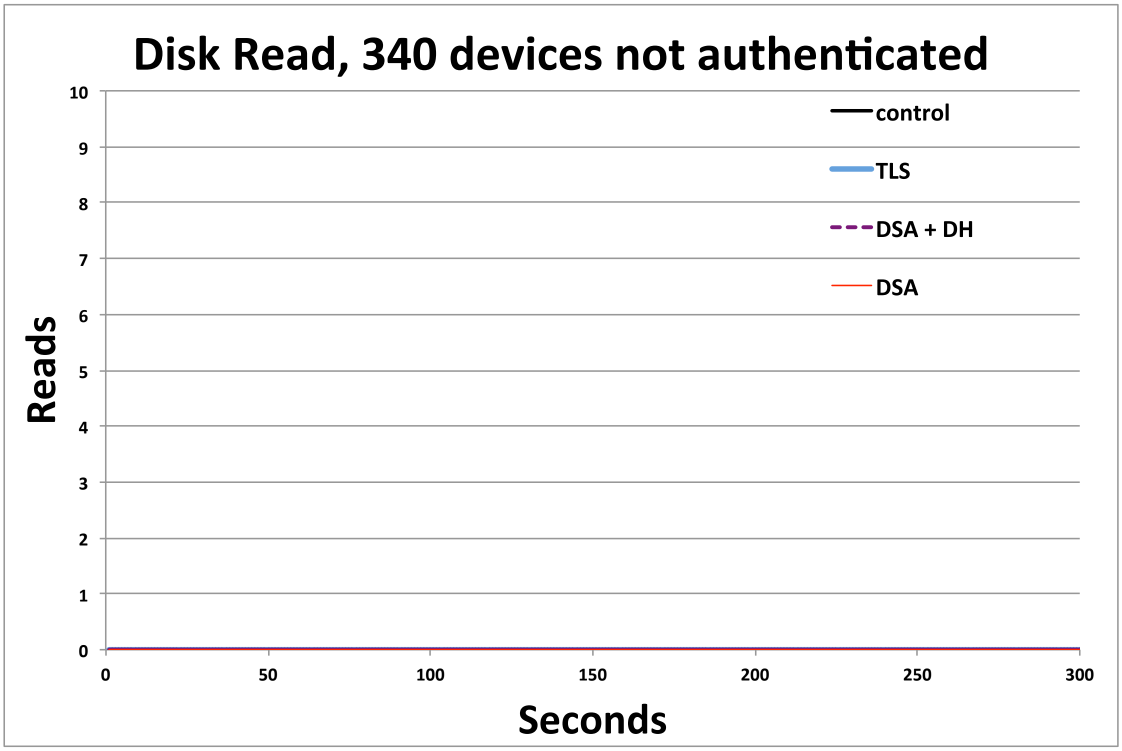
Authentication Framework Performance Data

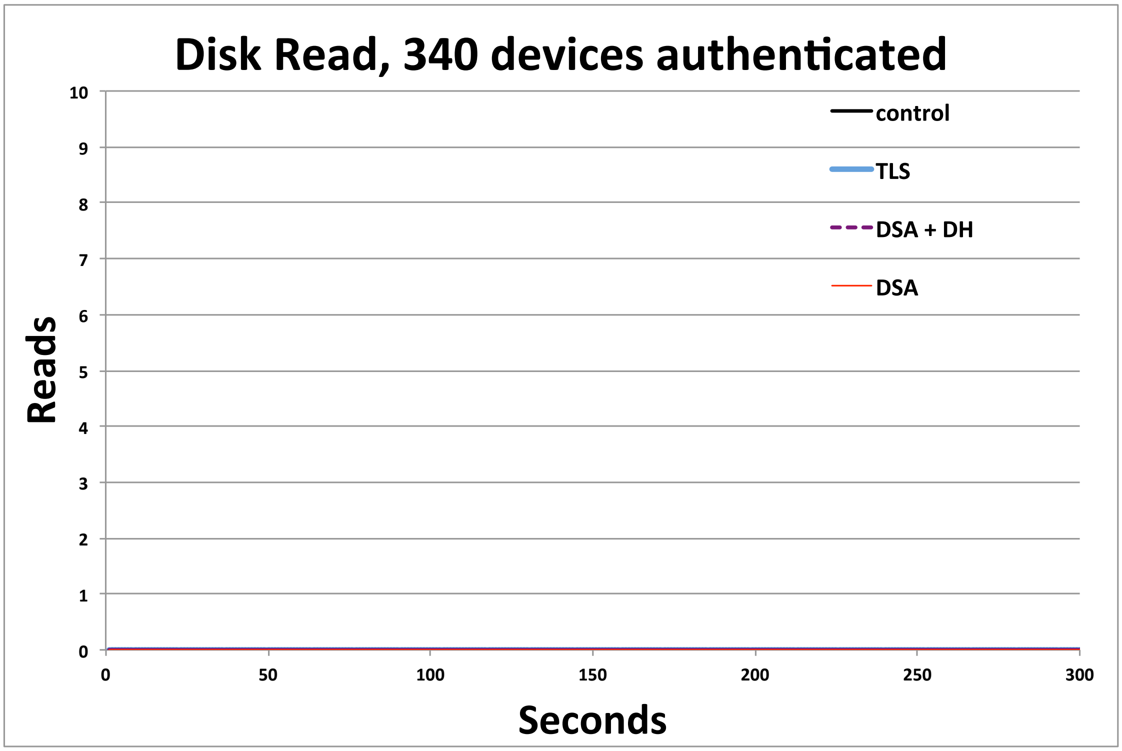


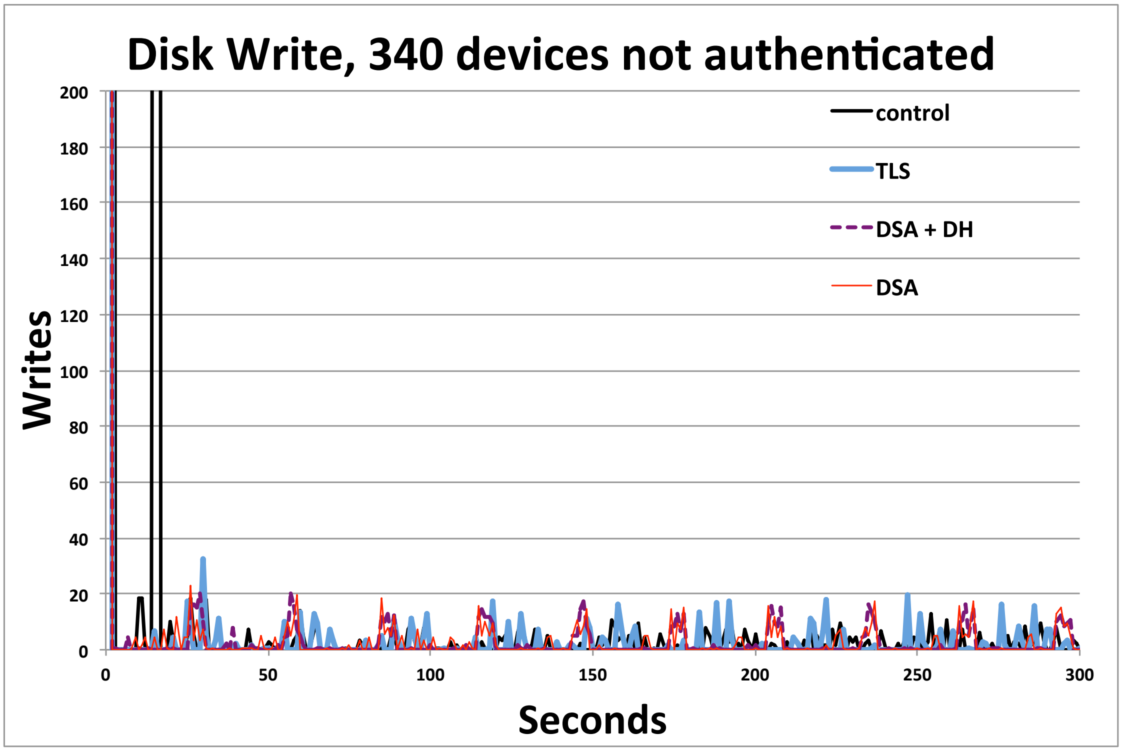

CPU usage without authentication
% CPU utilization
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
50
100
150
200
250
300
Time since system start
control
TLS
DSA + DH
DSA

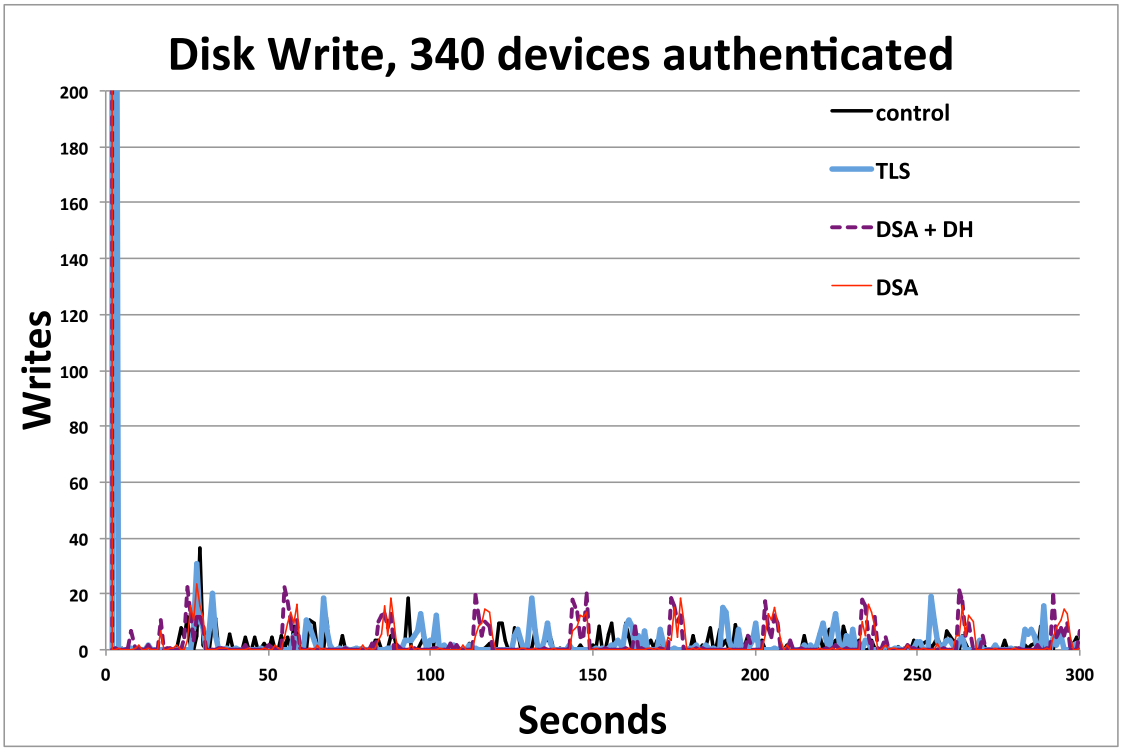

Memory usage without authentication
% Memory utilization
10%
9%
8%
7%
6%
5%
4%
3%
2%
1%
0%
0
50
100
150
200
250
300
Time since system start
control
TLS
DSA + DH
DSA

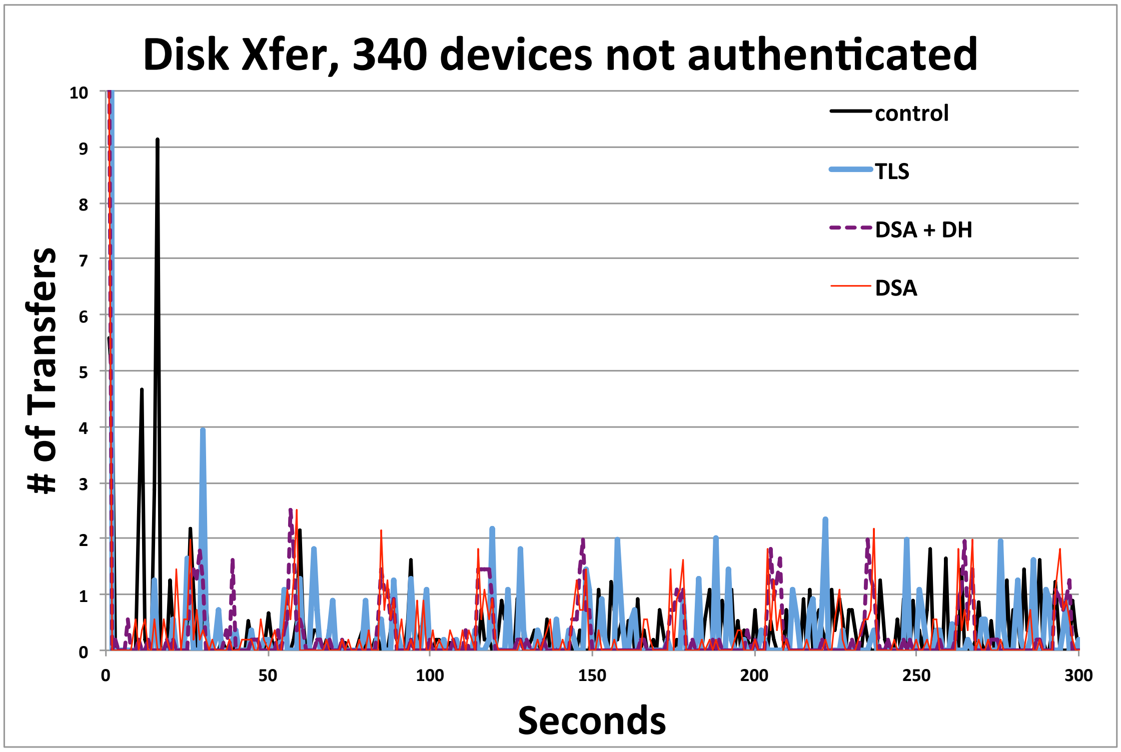

Memory usage with authentication
% Memory utilization
10%
9%
8%
7%
6%
5%
4%
3%
2%
1%
0%
0
50
100
150
200
250
300
Time since system start
control
TLS
DSA + DH
DSA

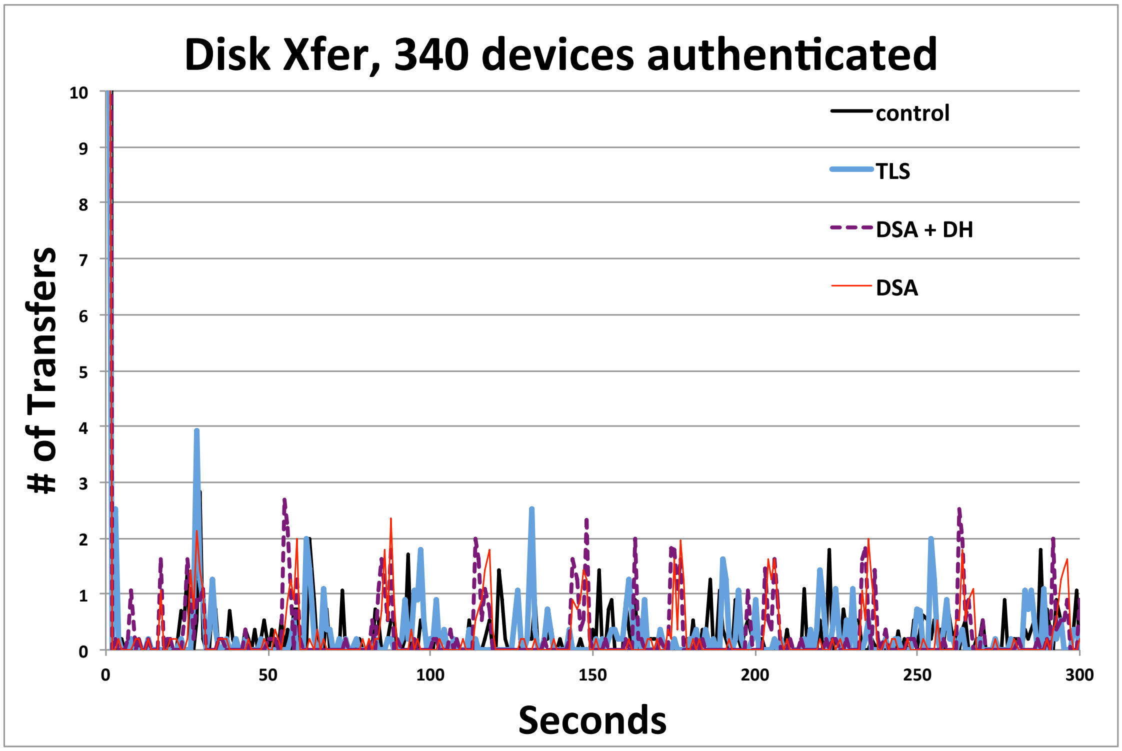

Disk Read, 340 devices not authenticated
Reads
10
9
8
7
6
5
4
3
2
1
0
0
50
100
150
200
250
300
Seconds
control
TLS
DSA + DH
DSA

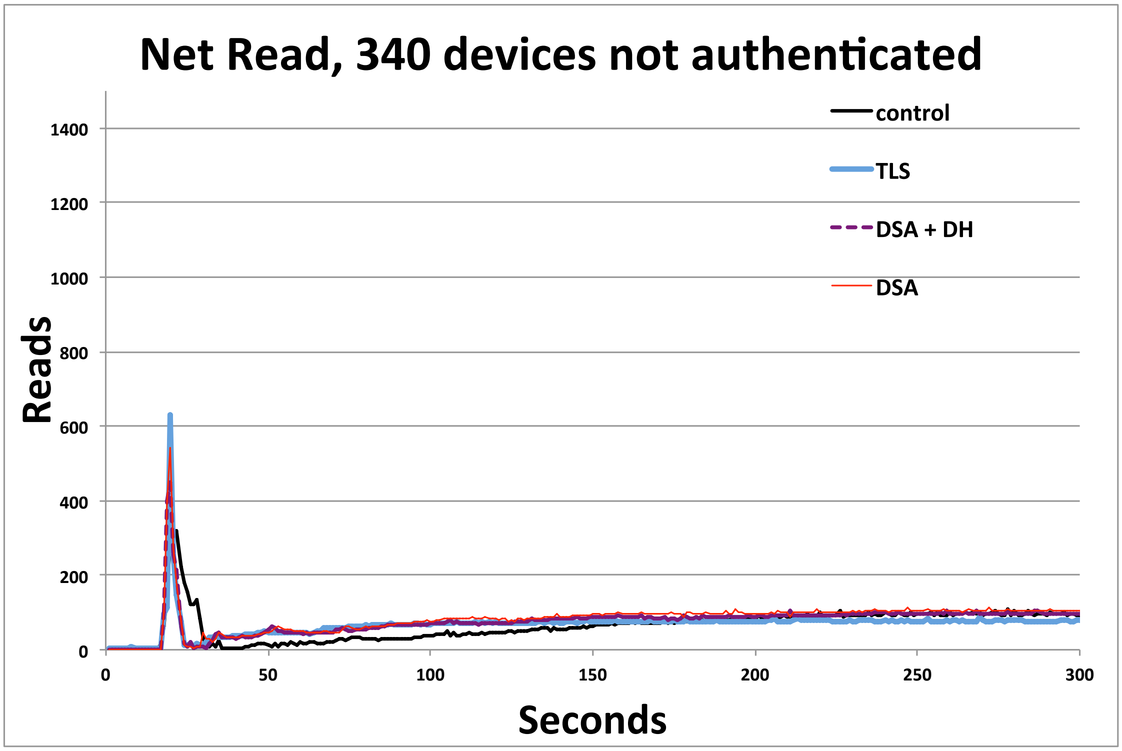

Disk Read, 340 devices authenticated
Reads
10
9
8
7
6
5
4
3
2
1
0
0
50
100
150
200
250
300
Seconds
control
TLS
DSA + DH
DSA

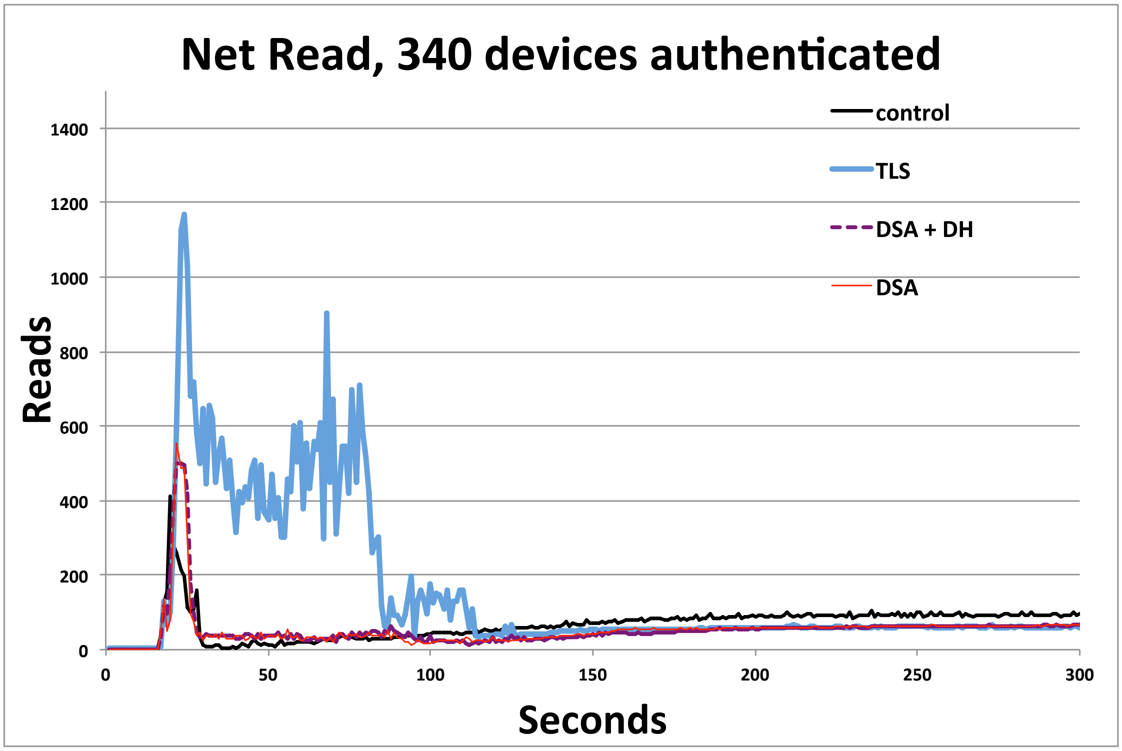

Disk Write, 340 devices not authenticated
Writes
200
180
160
140
120
100
80
60
40
20
0
0
50
100
150
200
250
300
Seconds
control
TLS
DSA + DH
DSA

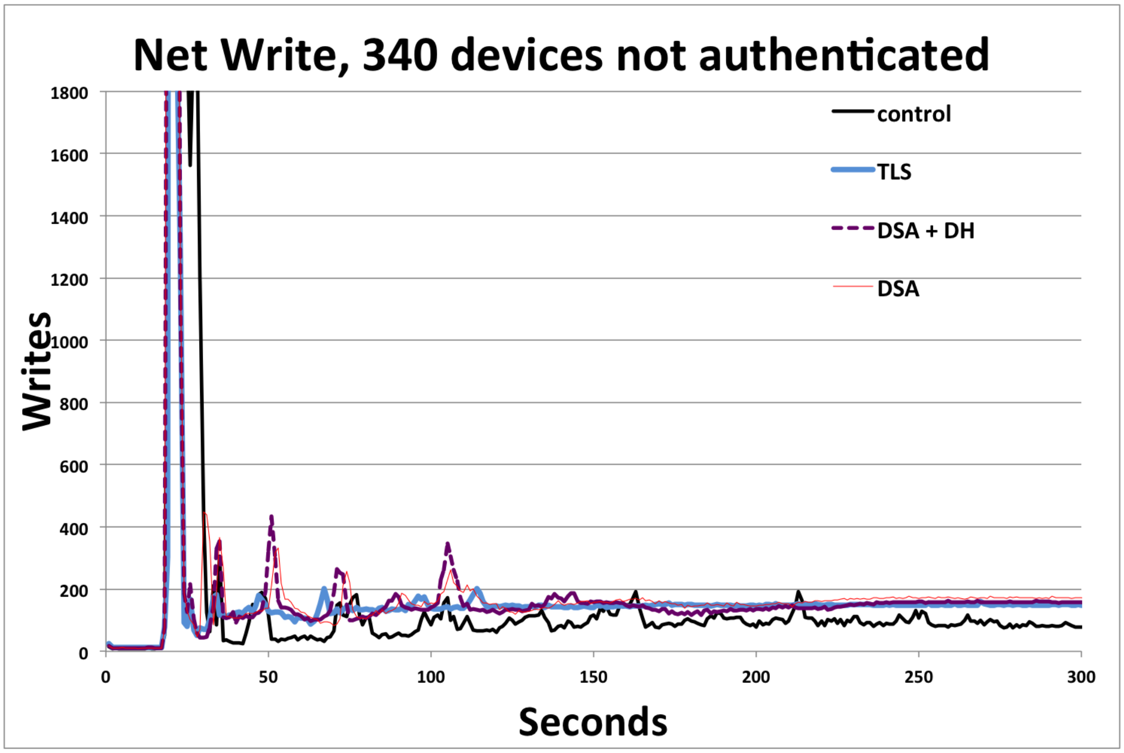

Disk Write, 340 devices authenticated
Writes
200
180
160
140
120
100
80
60
40
20
0
0
50
100
150
200
250
300
Seconds
control
TLS
DSA + DH
DSA

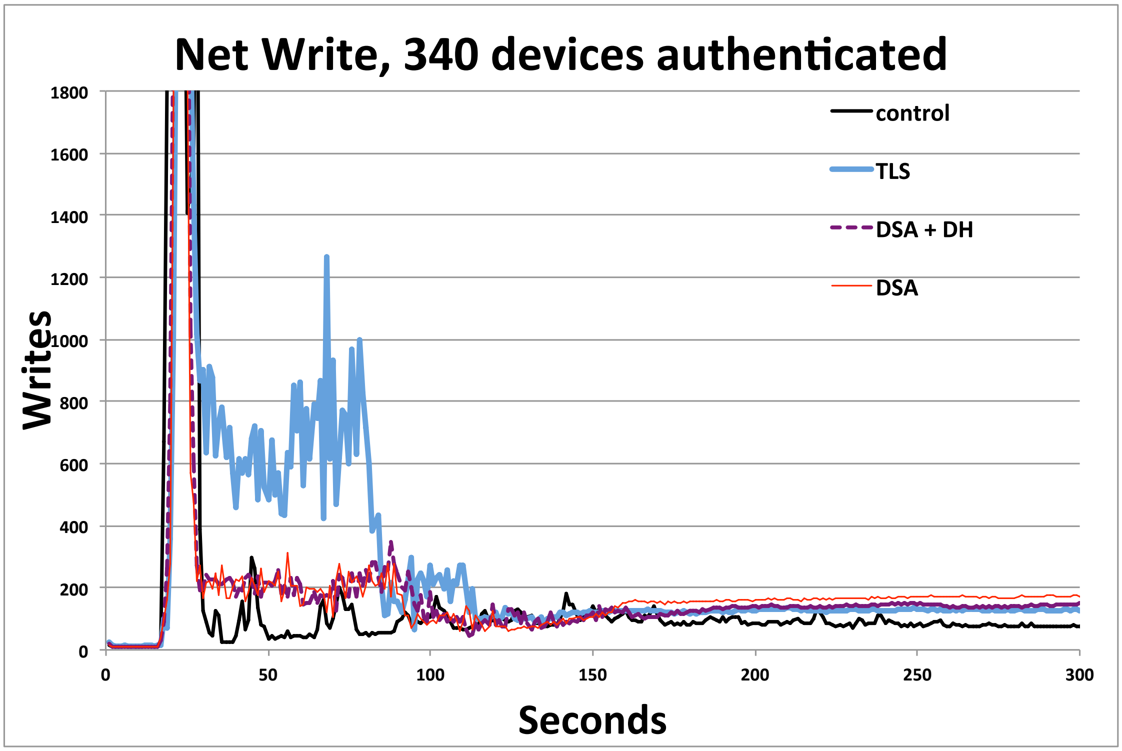

Disk Xfer, 340 devices not authenticated
# of Transfers
10
9
8
7
6
5
4
3
2
1
0
0
50
100
150
200
250
300
Seconds
control
TLS
DSA + DH
DSA

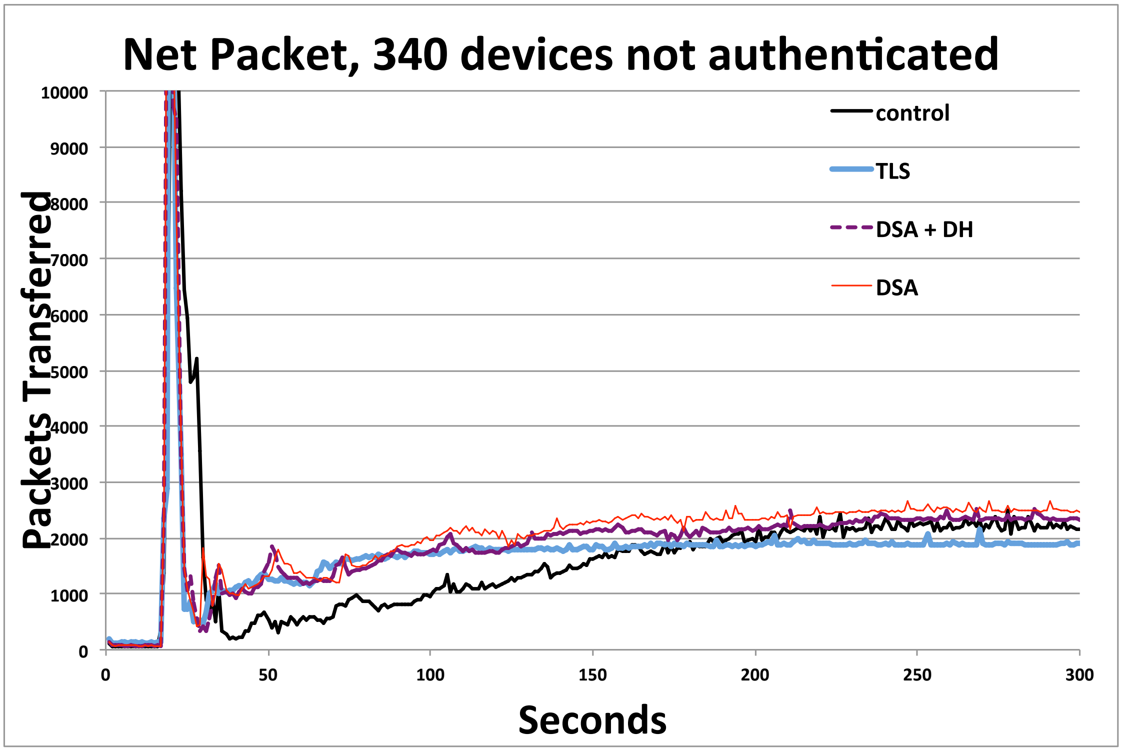

Disk Xfer, 340 devices authenticated
# of Transfers
10
9
8
7
6
5
4
3
2
1
0
0
50
100
150
200
250
300
Seconds
control
TLS
DSA + DH
DSA

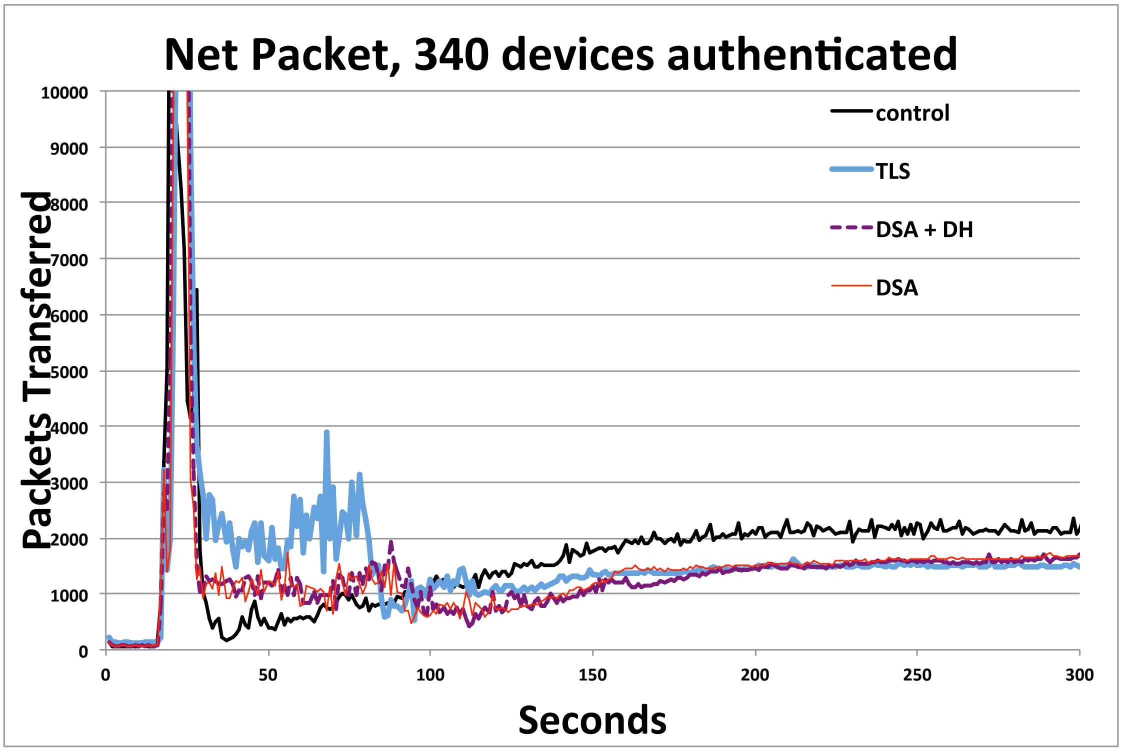

Net Read, 340 devices not authenticated
Reads
1400
1200
1000
800
600
400
200
0
0
50
100
150
200
250
300
Seconds
control
TLS
DSA + DH
DSA


Net Read, 340 devices authenticated
Reads
Seconds
0
200
400
600
800
1000
1200
1400
0
50
100
150
200
250
300
control
TLS
DSA + DH
DSA


Net Write, 340 devices not authenticated
Writes
Seconds
0
200
400
600
800
1000
1200
1400
1600
1800
0
50
100
150
200
250
300
control
TLS
DSA + DH
DSA


Net Write, 340 devices authenticated
Writes
1800
1600
1400
1200
1000
800
600
400
200
0
0
50
100
150
200
250
300
Seconds
control
TLS
DSA + DH
DSA


Net Packet, 340 devices not authenticated
Packets Transferred
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
50
100
150
200
250
300
Seconds
control
TLS
DSA + DH
DSA


Net Packet, 340 devices authenticated
Packets Transferred
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
50
100
150
200
250
300
Seconds
control
TLS
DSA + DH
DSA

Appendix B

Role Based Access Control System Use Cases

B.1 Use Cases

B.1.1 Normal Operation

Related System Goals: G1, G2, G3, G4

Primary Actor: Clinician

Precondition:

- MDCF is running

Postcondition:

- MDCF is running

Main Success Scenario:

1. Clinician logs in (B.1.3)

2. Clinician connects device (B.1.5)

3. Clinician launches app (bound to device) (B.1.7)

4. Clinician views data in app

5. Clinician issues command to device through app (B.1.9)

6. Clinician closes app (B.1.8)

7. Clinician disconnects device (B.1.6)

8. Clinician logs out (B.1.4)

B.1.2 Break Glass Mode Operation

Related System Goals: G5

Primary Actor: Clinician

Precondition:

- MDCF is running

- MDCF is not in Break Glass Mode

Postcondition:

- MDCF is running

- MDCF is in Break Glass Mode

Main Success Scenario:

1. Clinician logs in (B.1.3)

2. Clinician connects device

3. Clinician launches app (bound to device) (B.1.7)

4. Clinician views data in app

5. Clinician issues command to device through app(Failure) (B.2.5)
6. Clinician enters Break Glass Mode (B.1.10)
7. Clinician issues command to device (Break Glass) (B.1.16)
8. Clinician closes app (B.1.15)
9. Clinician disconnects device (B.1.13)
10. Administrator exits Break Glass Mode (B.1.11)

Alternate Scenario:

1. Clinician enters Break Glass Mode (B.1.10)
2. Clinician connects device (B.1.12)
3. Clinician launches app (bound to device) (B.1.14)
4. Clinician views data in app
5. Clinician issues command to device (Break Glass) (B.1.16)

B.1.3 User Log In

Related System Goals: G1, G2, G3, G4

Primary Actor: Clinician

Precondition:

- MDCF is running
- Clinician is not logged in

Postcondition:

- MDCF is running

- Clinician is logged in

Main Success Scenario:

1. Clinician enters credentials into Supervisor
2. Credentials transmitted to User Authentication module
3. User Authentication module sends log in confirmation to Supervisor
4. Supervisor starts user session for the Clinician

B.1.4 User Log Out

Related System Goals: G1, G2, G3, G4

Primary Actor: Clinician

Precondition:

- MDCF is running
- Clinician is logged in

Postcondition:

- MDCF is running
- Clinician is not logged in

Main Success Scenario: (without App permissions or roles)

1. Clinician issues logout command in Supervisor
2. All running Apps are closed (B.1.8)
3. Network Controller and Supervisor terminates user session for the Clinician.

Main Success Scenario: (App permissions and roles)

1. Clinician issues logout command in Supervisor
2. Apps continue running unless explicitly terminated
3. Supervisor terminates user session for the Clinician.

B.1.5 Device Connects (first time)

Related System Goals: G3, G8

Primary Actor: Clinician

Precondition:

- MDCF is running
- Device is not connected to MDCF
- Network Controller has no DML metadata cached for this device

Postcondition:

- MDCF is running
- Device is connected to MDCF
- Network Controller caches DML metadata for this device

Main Success Scenario:

1. Device is physically plugged into MDCF network switch or router
2. Device is powered on
3. Device runs through connection process (including authentication) with the Network Controller
4. Network Controller creates cached copies of the DML metadata and certificates for that device

5. RBAC Engine stores permissions and role-permission bindings for device
6. (Optional) permissions are automatically assigned to non-default roles based on local RBAC policy
7. Device is ready to use

B.1.6 Device Disconnects

Related System Goals: G3, G8

Primary Actor: Clinician

Precondition:

- MDCF is running
- Device is connected to the MDCF
- Device may be bound to one or more apps

Postcondition:

- MDCF is running
- Device is not connected to the MDCF
- Device is no longer bound to any apps

Main Success Scenario:

1. User issues command to disconnect Device within Supervisor Console
2. Supervisor console requests to perform disconnection Operation
3. RBAC Engine approves request to disconnect device
4. RBAC Engine forwards disconnection command to device manager
5. Device manager disconnects device

B.1.7 App Launch

Related System Goals: G3, G4

Primary Actor: Clinician

Precondition:

- MDCF is running
- Clinician is logged in
- Device is connected
- Device is compatible with App
- App is not running and is not bound to Device

Postcondition:

- MDCF is running
- Clinician is logged in
- Device is connected
- Device is compatible with App
- App is running and bound to Device

Main Success Scenario:

1. Clinician issues command in Supervisor to launch App and bind it to Device
2. App binding/launching request is sent to RBAC Engine
3. RBAC Engine verifies that Clinician has a Role that has the permission needed to launch App
4. RBAC Engine forwards command to other components to execute App launch process

5. App components are initialized
6. Supervisor displays App UI to Clinician

B.1.8 App Close

Related System Goals: G3, G4

Primary Actor: Clinician

Precondition:

- MDCF is running
- Clinician is logged in
- App is running
- Device is bound to App

Postcondition:

- MDCF is running
- Clinician is logged in
- App is not running
- Device is not bound to App

Main Success Scenario:

1. Clinician issues command to close App within Supervisor console
2. Supervisor requests permission to close App from RBAC Engine
3. RBAC Engine verifies Clinician has a Role with permission to close App, sends response to supervisor

4. App terminate message sent to Network Controller

5. App binding to device is removed

6. App UI is closed on Supervisor

B.1.9 Issue Command to Device

Related System Goals: G3

Primary Actor: MDCF App

Precondition:

- MDCF is running
- Device is connected
- App is running and bound to Device
- Clinician is logged in
- Clinician is assigned to Role
- Role has Permission
- Permission is for desired command on Device
- The command has not been executed

Postcondition:

- MDCF is running
- Device is connected
- App is running and bound to Device
- Command has been executed

Main Success Scenario:

1. Clinician indicates that a command should be issued to Device within App
2. App requests permission to issue command to Device
3. RBAC Engine verifies that Clinician has Role and Role has Permission
4. RBAC Engine approves request
5. RBAC Engine forwards command to Device
6. Device executes command

B.1.10 Enter Break Glass Mode

Related System Goals: G5

Primary Actor: Clinician

Precondition:

- MDCF is running
- MDCF is not in Break Glass mode

Postcondition:

- MDCF is running
- MDCF is in Break Glass mode

Main Success Scenario:

1. User issues command to enter Break Glass mode from Supervisor Console
2. Supervisor console sends message to RBAC Engine indicating Break Glass mode is active
3. Supervisor console visually indicates that Break Glass mode is active

B.1.11 Exit Break Glass Mode

Related System Goals: G5

Primary Actor: Administrator

Precondition:

- MDCF is running
- MDCF is in Break Glass mode
- Administrator is logged in

Postcondition:

- MDCF is running
- MDCF is not in Break Glass mode

Main Success Scenario:

1. Administrator issues command to exit Break Glass mode from Supervisor console
2. Supervisor console sends request to exit Break Glass mode to RBAC Engine
3. RBAC Engine approves request to exit Break Glass mode
4. RBAC Engine announces return to normal MDCF operation
5. (If App roles present) Apps launched during Break Glass operation continue to run, using default roles
6. (otherwise) Apps launched while MDCF was in Break Glass mode are terminated
7. Unknown Devices connected during Break Glass mode are disconnected
8. Supervisor Console removes visual indication of Break Glass mode

B.1.12 (Break Glass) Device Connect (Unknown Device)

Related System Goals: G3, G5

Primary Actor: Clinician

Precondition:

- MDCF is running
- MDCF is in Break Glass mode
- Device is not connected to MDCF

Postcondition:

- MDCF is running
- MDCF is in Break Glass mode
- Device is connected to MDCF

Main Success Scenario:

1. Device transmits DML to Network Controller
2. DML is used for matching, other essential purposes, but no access control actions are taken in Break Glass mode
3. Device finishes connection process, is ready for use

B.1.13 (Break Glass) Device Disconnect

Related System Goals: G3, G5

Primary Actor: Clinician

Precondition:

- MDCF is running

- MDCF is in Break Glass mode
- Device is connected to the MDCF
- Device may be bound to one or more apps

Postcondition:

- MDCF is running
- MDCF is in Break Glass mode
- Device is not connected to the MDCF
- Device is no longer bound to any apps

Main Success Scenario:

1. User issues command to disconnect Device within Supervisor Console
2. Supervisor console requests to perform disconnection operation
3. RBAC Engine forwards disconnection command to Device Manager
4. Device Manager disconnects Device

B.1.14 (Break Glass) App Launch

Related System Goals: G5, G6

Primary Actor: Clinician

Precondition:

- MDCF is running
- MDCF is in Break Glass mode
- Device is connected

- Device is compatible with App
- App is not running and is not bound to Device

Postcondition:

- MDCF is running
- MDCF is in Break Glass mode
- Device is connected
- Device is compatible with App
- App is running and bound to Device

Main Success Scenario:

1. Clinician issues command in Supervisor to launch App and bind it to Device
2. App binding/launching request is sent to RBAC Engine
3. RBAC Engine forwards command to other components to execute App launch process
4. App components are initialized
5. Supervisor displays App UI to Clinician

B.1.15 (Break Glass) App Close

Related System Goals: G5, G6

Primary Actor: Clinician

Precondition:

- MDCF is running
- MDCF is in Break Glass mode

- App is running
- Device is bound to App

Postcondition:

- MDCF is running
- MDCF is in Break Glass mode
- App is not running
- Device is not bound to App

Main Success Scenario:

1. Clinician issues command to close App within Supervisor console
2. Supervisor requests permission to close App from RBAC Engine
3. RBAC Engine forwards App terminate command to Network Controller components
4. App binding to device is removed
5. App UI is closed on Supervisor

B.1.16 (Break Glass) Issue Device Command

Related System Goals: G3, G5, G6

Primary Actor: Clinician

Precondition:

- MDCF is running
- MDCF is in Break Glass mode
- Device is connected

- App is running and bound to Device
- The command has not been executed

Postcondition:

- MDCF is running
- MDCF is in Break Glass mode
- Device is connected
- App is running and bound to Device
- Command has been executed

Main Success Scenario:

1. Clinician indicates that a command should be issued to Device within App
2. App requests permission to issue command to Device
3. RBAC Engine forwards command to Device
4. Device executes command

B.1.17 Create Role

Related System Goals: G8

Primary Actor: Administrator

Precondition:

- MDCF is running
- Role is not in access control policy

Postcondition:

- MDCF is running
- Role is in access control policy

Main Success Scenario:

1. Administrator edits MDCF security policy to add a new role
2. Administrator saves changes, and issues command in Administrative Console to reload policy
3. Administrative console sends request to reload policy to RBAC Engine
4. RBAC engine approves request, and forwards command to relevant parts of Network Controller
5. MDCF NC loads new policy containing the additional role

B.1.18 Delete Role

Related System Goals: G8

Primary Actor: Administrator

Precondition:

- MDCF is running
- Role is in access control policy

Postcondition:

- MDCF is running
- Role is not in access control policy

Main Success Scenario:

1. Administrator edits MDCF security policy to remove a role
2. Administrator saves changes, and issues command in Administrative Console to reload policy
3. Administrative console sends request to reload policy to RBAC Engine
4. RBAC engine approves request, and forwards command to relevant parts of Network Controller
5. MDCF NC loads new policy which does not have the role
6. Role is removed from all Users

B.1.19 Add Role To User

Related System Goals: G8

Primary Actor: Administrator

Precondition:

- MDCF is running
- Administrator is logged in
- Role is an existing role in the MDCF
- User is an existing user in the MDCF
- User is not associated with Role

Postcondition:

- MDCF is running
- Administrator is logged in

- Role is an existing role in the MDCF
- User is an existing user in the MDCF
- User is associated with Role

Main Success Scenario:

1. Administrator opens Administrative Console
2. Administrator edits configuration for user in Administrative Console
3. Administrator adds Role to User in Administrative Console
4. Administrator confirms changes
5. Administrative Console broadcasts change to update cache on Supervisor (if applicable)

B.1.20 Remove Role From User

Related System Goals: G8

Primary Actor: Administrator

Precondition:

- MDCF is running
- Administrator is logged in
- Role is an existing role in the MDCF
- User is an existing user in the MDCF
- User is associated with Role

Postcondition:

- MDCF is running
- Administrator is logged in
- Role is an existing role in the MDCF
- User is an existing user in the MDCF
- User is not associated with Role

Main Success Scenario:

1. Administrator opens Administrative Console
2. Administrator removes Role from User in Administrative Console
3. Administrator confirms changes
4. Administrative Console broadcasts change to update cache on Supervisor (if applicable)

B.1.21 Add Permission to Role

Related System Goals: G8

Primary Actor: Administrator

Precondition:

- MDCF is running
- Administrator is logged in
- Role is an existing role in the MDCF
- User is an existing user in the MDCF
- User is associated with Role

Postcondition:

- MDCF is running
- Administrator is logged in
- Role is an existing role in the MDCF
- User is an existing user in the MDCF
- User is not associated with Role

Main Success Scenario:

1. Administrator opens Administrative Console
2. Administrator removes Role from User in Administrative Console
3. Administrator confirms changes
4. Administrative Console broadcasts change to update cache on Supervisor (if applicable)

B.1.22 Remove Permission from Role

Related System Goals: G8

Primary Actor: Administrator

Precondition:

- MDCF is running
- Administrator is logged in
- Role is an existing role in the MDCF
- User is an existing user in the MDCF

- User is associated with Role

Postcondition:

- MDCF is running
- Administrator is logged in
- Role is an existing role in the MDCF
- User is an existing user in the MDCF
- User is not associated with Role

Main Success Scenario:

1. Administrator opens Administrative Console
2. Administrator removes Role from User in Administrative Console
3. Administrator confirms changes
4. Administrative Console broadcasts change to update cache on Supervisor (if applicable)

B.2 Exception Cases

B.2.1 User Log In (Failure)

Related System Goals: G1, G2, G3, G4

Primary Actor: Clinician

Precondition:

- MDCF is running
- Clinician is not logged in

Postcondition:

- MDCF is running
- Clinician is not logged in

Main Success Scenario:

1. Clinician enters credentials into Supervisor
2. Credentials transmitted to Network Controller
3. User Authentication module can not verify credentials
4. Supervisor informs Clinician of failed log in attempt

B.2.2 App Launch (Failure)

Related System Goals: G1, G2, G3, G4, G7

Primary Actor: Clinician

Precondition:

- MDCF is running
- Clinician is logged in
- Device is connected
- Device is compatible with App
- App is not running and is not bound to Device

Postcondition:

- MDCF is running
- Clinician is logged in

- Device is connected
- Device is compatible with App
- App is running and bound to Device

Main Success Scenario:

1. Clinician issues command in Supervisor to launch App and bind it to Device
2. App binding/launching request is sent to RBAC Engine
3. RBAC Engine verifies that Clinician has a Role that has the permission needed to launch App
4. RBAC Engine forwards command to other components to execute App launch process
5. App components are initialized
6. Supervisor displays App UI to Clinician

B.2.3 App Close (Failure)

Related System Goals: G1, G2, G3, G4, G7

Primary Actor: Clinician

Precondition:

- MDCF is running
- Clinician is logged in
- App is running
- Device is bound to App

Postcondition:

- MDCF is running
- Clinician is logged in
- App is running
- Device is bound to App

Main Success Scenario:

1. Clinician issues command to close App within Supervisor console
2. Supervisor requests permission to close App from RBAC Engine
3. RBAC Engine rejects request to terminate app (user doesn't have permission)
4. Supervisor informs user that they do not have permission to close the app

B.2.4 Device Disconnects (Failure)

Related System Goals: G6, G7

Primary Actor: Clinician

Precondition:

- MDCF is running
- Device is connected to the MDCF

Postcondition:

- MDCF is running
- Device is connected to the MDCF

Main Success Scenario:

1. User issues command to disconnect Device within Supervisor Console

2. Supervisor console requests to perform disconnection Operation
3. RBAC Engine denies request to disconnect Device due to missing permission
4. Supervisor notifies user that they do not have permission to disconnect Device

B.2.5 Issue Command To Device (Failure)

Related System Goals: G2, G4, G6

Primary Actor: Clinician

Precondition:

- MDCF is running
- Device is connected
- App is running and bound to Device
- The command has not been executed

Postcondition:

- MDCF is running
- Device is connected
- App is running and bound to Device
- Command has not been executed

Main Success Scenario:

1. Clinician indicates that a command should be issued to Device within App
2. App requests permission to issue command to Device
3. RBAC Engine denies request due to insufficient permission
4. Supervisor console notifies Clinician of insufficient permission to issue command

B.2.6 Exit Break Glass Mode (Failure)

Related System Goals: G5, G8

Primary Actor: Administrator

Precondition:

- MDCF is running
- MDCF is in Break Glass mode

Postcondition:

- MDCF is running
- MDCF is in Break Glass mode

Main Success Scenario:

1. User issues command to exit Break Glass mode from Supervisor console
2. Supervisor console sends request to exit Break Glass mode to RBAC Engine
3. RBAC Engine denies request to exit Break Glass mode
4. RBAC Engine sends notification of denial to Supervisor console
5. Supervisor Console visually notifies User that they can not exit break glass mode

B.3 External Entities

The MDCF external entity is defined to specify environmental assumptions that span more than one external entity.

B.3.1 MDCF

- **Channel Confidentiality, Integrity, Authenticity:** Channels between the Supervisor and Network Controller, Apps and Network Controller, and Devices and Network Controller are assumed to have mechanisms in place providing confidentiality, integrity, and authenticity of the data transmitted.

  Rationale: This is necessary to prevent eavesdropping, also in combination with message integrity/authenticity mechanisms, this prevents circumvention of access control mechanisms from forged messages and replay attacks.

B.3.2 DML Metadata

- **DML Access Control Annotations:** These requirements assume the existence of Access Control information being embedded in the DML metadata for all devices. This information includes all possible permissions for a device, with default permission assignments to universally pre-defined roles.

  Rationale: We need a standard way of storing and transmitting access control information for devices. Each device has DML metadata, making it an ideal location for access control information.

- **DML Access Control Annotation Parsing:** We assume that there is a component in the MDCF which parses access control data from DML and converts it into a format usable by the RBAC Engine.

  Rationale: In order to use the DML Access Control Annotations for devices, we need to be able to read and use that data, thus we need a parser.

B.3.3 User management system

- **User Database:** We assume there is a Database of users in the system, accessible by the RBAC Engine.

Rationale: We need a data structure that stores users, so that we can then do role assignment to those users.

- **User Authentication:** We assume that mechanisms are in place for authenticating users.

  Rationale: This is a standard feature of any multi-user system. Access control depends on a reliable user authentication system. Without it, there is no guarantee that the person associated with a user account is the intended person.

- **User Sessions:** We assume that the User Management system keeps track of which user(s) are logged in at any given moment.

  Rationale: We want to be sure of which user is using the MDCF, so that we can make correct access control decisions based on the roles and permissions associated with that user.

B.3.4 Supervisor App

- **Supervisor App DML:** We assume that there exists DML or some equivalent form of metadata for Apps, this can contain access control information for specific apps.

  Rationale: Access control for apps themselves will require the device metadata to specify its permissions.

B.3.5 Supervisor Console/Operator Interface

- **Supervisor Console Locking:** We assume that there is some mechanism that prevents individuals other than the clinician currently logged into to the MDCF from using the clinician's user account when that clinician is not actively using the MDCF Supervisor console.

  Rationale: We want to be sure that an unauthorized person can't easily control apps and devices using the clinician's already logged in account. (e.g. The clinician logs in, then steps out of the room briefly causing the Supervisor console to automatically lock until that clinician returns or someone else logs in)

- **System Status Indication:** We assume that the Supervisor console has some means of displaying feedback from the RBAC Engine to a user.
  Rationale: We need to be able to inform users when an operation is prevented by the RBAC Engine, and when the system in in Break Glass Mode.

- **Break Glass Controls:** We assume that there is a way, through the Supervisor console to enter and exit Break Glass mode.
  Rationale: Clinicians will primarily interact with the MDCF through the Supervisor console, making it the convenient and logical location for the Break Glass controls.

- **Administrative Console:** We assume the existence of an Administrative Console, either as a part of the Supervisor Console, or as a separate entity. The Administrative console allows a system administrator to change MDCF configuration and security policy settings.
  Rationale: An Administrative console will allow administrators to define security policy without complicating the rest of the Clinician-facing Supervisor console.

B.4 Requirements

B.4.1 RBAC System Function

The RBAC Engine performs two functions. The first is to facilitate access control; grant or deny the ability to carry out critical operations in the MDCF. The second is to activate/deactivate the Break Glass mode of the MDCF.

The high-level requirements of the Access Control System Function are as follows:

1. **Requirement Title:** The RBAC system shall store and maintain roles

Rationale: Role based access control relies on roles, thus the system needs a means of storing roles.

2. **Requirement Title:** The RBAC system shall store and maintain permissions

   Rationale: In RBAC, nothing can be done without a permission, thus the system needs to track permissions.

3. **Requirement Title:** The RBAC system shall store and maintain user assignments

   Rationale: Users need to be assigned to roles in order to use the permissions for that role, user assignments are associations between users and roles.

4. **Requirement Title:** The RBAC system shall store and maintain permission assignments

   Rationale: Roles need to have permissions assigned to them, this is a permission assignment.

5. **Requirement Title:** The RBAC system shall include default roles

   Rationale: A deployed system may eventually have custom roles, however in order for the MDCF to work in a plug-and-play system, there needs to be a default set of roles that are universally present in any MDCF system or device.

6. **Requirement Title:** The RBAC system shall allow an Administrator to create custom roles

   Rationale: Clinical environments where the MDCF might be deployed will all have different organizational structures and operating procedures. Administrators need to be able to create custom roles to match the needs of their organization.

7. **Requirement Title:** The RBAC system shall prevent a command from being sent to a device unless the sender has a role with permission to send the command

   Rationale: Control of a device needs to be restricted to protect patient safety and enforce whatever access control policies a care facility has set.

8. **Requirement Title:** The RBAC system shall prevent a user from launching an App unless the user has a role with permission to launch the App

   Rationale: Launching an app allows it to have some amount of access to a device's data, and potentially control over that device. Thus, restricting the launching of apps may be necessary depending on a care facility's needs.

9. **Requirement Title:** The RBAC system shall prevent a User from gaining access to an App unless the user has a role with permission to access the App

   Rationale: Apps can provide access to sensitive patient data, or may allow control of a device, thus restricting access to running apps may be necessary.

10. **Requirement Title:** The RBAC system shall prevent a user from closing an App unless the user has a role with permission to close the App

    Rationale: Apps may potentially be responsible for ongoing or repetitive tasks that are medically relevant, thus closing an app can be a critical operation that should be restricted.

11. **Requirement Title:** The RBAC system shall prevent a user from disconnecting a Device unless the user has a role with permission to disconnect the device

    Rationale: Devices may be associated to apps and executing some sort of medically relevant task. Furthermore, this may prevent someone who is not physically present from disconnecting devices.

12. **Requirement Title:** The RBAC system shall provide a means for managing roles, permissions, role assignments, and permission assignments

    Rationale: In order for RBAC to be effective, it needs to be configurable so that any care facility or institution using the MDCF can define access control policies to fit their individual needs.

13. **Requirement Title:** The RBAC system shall automatically gather Access Control data from DML for devices and apps

Rationale: Devices and apps are likely to have unique permissions which need to be recognized and stored by the RBAC engine. These permissions as well as default permission assignments to the default roles should be stored in the DML for a device or app.

B.4.2 Break Glass System Function

The Break Glass Function bypasses the Access Control function, allowing clinicians to quickly provide care in an emergency situation that deviates from what is typical in the environment the MDCF is deployed in. Excessive use of the Break Glass feature may an indication that the security policies normally in place are overly restrictive.

The high-level requirements of the Break Glass System Function are as follows:

1. **Requirement Title:** The Break Glass system shall allow users to activate Break Glass mode

   Rationale: Break Glass mode can not be used if it cannot be activated.

2. **Requirement Title:** The Break Glass system shall allow users to launch any app with a compatible device while in Break Glass mode

   Rationale: In order to deal with an emergency/exceptional event, clinicians should be able to launch apps as needed to deliver care to the patient, regardless of their normal roles.

3. **Requirement Title:** The Break Glass system shall allow users to view and access all features of running apps while in Break Glass mode

   Rationale: In order to deal with an emergency/exceptional event, clinicians should be able to view and control apps as needed to deliver care to the patient, regardless of their normal roles.

4. **Requirement Title:** The Break Glass system shall allow full, unrestricted control of devices while in Break Glass mode

Rationale: In order to deal with an emergency/exceptional event, clinicians should be able to control devices as needed to deliver care to the patient, regardless of their normal roles.

5. **Requirement Title:** The Break Glass system shall broadcast a notification when Break Glass mode is entered or exited

   Rationale: The Break Glass system part of the RBAC Engine needs to notify other components of the MDCF that a Break Glass scenario is occurring so that the other components may execute whatever special break glass functions they are responsible for.