Abstract - This paper presents SPARTA (Secure, Policy-driven, ARchitecture for content disTribution and storAge), a secure, policy-driven architecture for enterprise applications operating in centralized wireless environments. SPARTA supports end-to-end client authentication, data integrity and confidentiality. The security services provided by SPARTA are customized and controlled by easily configurable security policies which specify several security-related attributes, classify network and locally-stored data based on sensitivity and content, and provide an abstraction for the communication and messaging between the client and the server. In addition, SPARTA provides a standard Application Programming Interface (API) that conceals to a great extent the complexity of security operations and programming from the application. SPARTA was designed in a platform-neutral manner and can be implemented on a wide range of wireless clients ranging from low-end platforms such as the Java 2 Mobile Edition/Connected Limited Device Configuration (J2ME/CLDC) on limited-memory mobile devices to Personal Java and the .Net Compact Framework on PDAs. On the server side, SPARTA can be implemented on any of the available enterprise server platforms. A sample implementation of SPARTA was developed for J2ME on the client-side and Java 2 Enterprise Edition (J2EE) on the server-side.

Keywords - security, wireless security, mobile commerce, policy-driven security, customizable security, AES

1. Introduction

Although the adoption of mobile commerce (m-commerce) services is showing some acceleration, great concerns are raised about the security of the sensitive data on the wireless device and over the wireless links where data confidentiality and integrity are potentially compromised by unauthorized access. Wireless applications have special and unique requirements compared to "wired" Internet applications: usually, these applications operate over low bandwidth networks with high latency and frequent disconnections using devices that vary greatly in capabilities and resources. For this reason, the protocols used in securing wireless enterprise applications have to be designed specifically for operation in wireless environments and must address the needs and requirements of a huge variety of devices which are, in majority, severely constrained in terms of processor speed, memory resources, and storage capacity. This diversity makes the implementation of a unique security standard to encompass the whole device range infeasible. A least-common denominator security standard that targets devices with limited memory and slow processors would be unfair for powerful devices and would not meet their security requirements, and in the same sense, a security standard that addresses high-end devices would neither fit nor perform efficiently on limited-resource devices. What is needed, therefore, is a security protocol that can be customized and configured to perform the security operations flexibly, taking into consideration the memory capabilities and the processing power of the client device, the wireless network latency, and the specific requirements of the enterprise application. Moreover, this protocol must be extensible, scalable and capable of evolving to meet new challenges and to adapt to new application requirements.

In this paper, we present SPARTA, a scalable, extensible, and customizable security architecture for wireless enterprise applications operating in a centralized environment. By a centralized wireless environment we mean a wireless network architecture following the client/server communication model where wireless nodes or clients connect to and use the resources of one or more central enterprise servers. SPARTA is specifically designed for operation in wireless environments where it supports authentication, data confidentiality and integrity security services for wireless enterprise systems. It is based on an easily configurable network and storage security policies that control several security-related attributes, provide a generic representation of the client/server communications, and specify a flexible and fine-grained encryption methodology that secures network and client-side local data based on content and sensitivity. SPARTA also provides its users the ability to transmit only the encrypted data part over the wireless links and to embed the plain text part statically in the network security policy. This leads to a reduction in network traffic and contributes to decreasing the number of encryption operations and controlling their level which results in great flexibility and an overall performance improvement. Moreover, SPARTA provides a standard API that hides the complexity of enterprise security programming from the application developer and helps in increasing productivity and delivering more reliable and secure applications. SPARTA is designed in a platform-neutral manner and
can be implemented on a wide range of wireless platforms and enterprise servers.

SPARTA is an extension of SPECSA [12], a policy-driven security architecture for wireless enterprise applications. SPARTA extends SPECSA by supporting customizable security services for application data stored in the on-device persistent storage. In SPARTA, the local data will be secured based on a storage security policy, functionally similar to the network security policy supported by SPECSA.

The rest of the paper is organized as follows. In Section 2, we give a brief review of related work dealing with wireless security. In Section 3, we discuss the design and architecture of SPARTA. This will include an overview of the different components comprising the architecture on the client and the server sides. Section 4 provides an overview of a complete implementation of SPARTA using J2ME and J2EE. A performance analysis of the implementation is presented in Section 5, followed by conclusions in Section 6.

2. Related Work

This section provides an overview of current solutions dealing with the security of wireless networks and applications and examines the degree to which the solutions are effective in securing wireless systems. It should be noted that none of the security protocols discussed in this section deals with the storage security of local data on the client device. Usually this service is provided explicitly by encrypting the whole local store contents. This strategy, in addition of being inflexible, wastes system resources by performing unnecessary encryption operations on limited-resource wireless devices.

KSSL: KSSL [4] is a light-weight, client-side implementation of SSL version 3.0. In spite of the advantage provided by KSSL in securing m-commerce applications, some comments are worth mentioning here. First, KSSL’s performance is considered unacceptable when the client needs to communicate with different servers or when browsing sensitive content. This is due to the fact that in such cases the full handshake SSL operation needs to be carried out every time the client connects to a new web server. Performing an abbreviated handshake and reusing the master secret repeatedly in generating the symmetric ciphering keys raises some concerns about the security of this master key on the wireless device which can be easily snooped or stolen. The second comment about the operation of SSL is its non-differentiation when performing the encryption operations on the wireless device. In other words, SSL indiscriminately encrypts all the network data with the same encryption strength without regard to its type or sensitivity. This can be unnecessary or even undesirable for some wireless applications.

SPARTA has given these issues a great attention. SPARTA supports a secure key management mechanism (see Section 3.4.1) where all the security-related attributes and ciphering keys are stored encrypted on the client device. A ‘one size fits all’ solution such as KSSL is not feasible in the diverse wireless world where no assumption can be made about the capability and performance of wireless devices or about the operation of applications on wireless networks. It is only the user and the service provider who can determine the actual needs and requirements of their application. For this reason, the security operations are based on an externally customizable and differential security policy which groups network data according to sensitivity and content, and takes into consideration the processing power and memory capabilities of the wireless device. This flexibility ensures efficient operation of the same application on a wide-range of devices and wireless networks.

Tiny SESAME: Tiny SESAME [9] is a light-weight implementation based on the Secure European System for Applications in a Multi-vendor Environment (SESAME) architecture [8]. SESAME is designed for operation in distributed systems where it provides access control, authentication, and data confidentiality and integrity. It supports the Kerberos authentication mechanism and extends it with additional services such as asymmetric cryptography based on public-key technology. Two main reasons make Tiny SESAME unsuitable for operation in wireless environments. First, SESAME regularly employs public-key/Kerberos operations that require intensive CPU and memory resources which lead to unacceptable response times in wireless applications [2]. This is evident in Tiny SESAME’s poor performance results presented in [9]. Second, the component-based Tiny SESAME architecture where resources are loaded dynamically, increases network traffic, and raises significant security risks on low-end wireless platforms that lack the standard security verification and access control mechanisms for controlling the operation of dynamically loaded resources. The absence of such security mechanisms is due to their complex and memory-consuming algorithms. This makes the implementation of Tiny SESAME only possible on high-end devices.

SPARTA is specifically designed for operation in restricted environments. It uses efficient symmetric-key operations with fast algorithms and all the architecture components are designed in a platform-neutral manner without assuming any special capabilities in their implementation, which ensures efficient operation on low end as well as high-end mobile devices.

XML-based Security: Some solutions proposed providing end-to-end security by transferring secure Extensible Markup Language (XML) documents between clients and servers. Such proposals relied on the several XML security protocols [10, 11] that have been proposed to support communication data security in XML applications. However, such proposals remained unimplemented due to the lack of XML support in dominating wireless platforms, such as J2ME/CLDC 1.0. This lack of support is due to the limited string functions in such platforms, which reduce the efficiency of XML parsing. In addition, the tagged structure of XML [13] makes it a rather heavy language for the limited wireless bandwidths, and imposes extra network traffic and overhead that is not necessary in a wireless environment. Add to this the fact that on
Today's wireless networks, latency is usually more of an issue than data transfer rate, so we will notice the larger message size of XML content versus a binary format. In SPARTA, the format of the network data is implementation dependent and there is a clear and clean separation between the data format used to transport the data between the communicating parties and the implementation of the security operations. This makes the solution applicable to a range of data formats without being bound to a single one, and allows for the effort to be concentrated on assuring the security of the solution and guaranteeing its optimal performance without making any assumptions about the format used in transporting the network data.

**J2ME End-to-End Security (JEES) and other miscellaneous security solutions:** JEES [1] is an end-to-end application-layer security solution for wireless applications using J2ME. The main limitation in JEES is that it is not easily extensible. This is due to the fact that all the security code is present in hard coded form in the application, thus adding new security operations requires the recompilation and redeployment of the application. In addition this approach requires the programmer to have the appropriate knowledge in network security concepts and in robust programming skills.

Some patterns that deal with local storage security are PKCS #1, PKCS #11, and PKCS #12. These are public-key cryptography standards published by RSA Data Security, Inc. [15]. They rely on the use of public-key cryptography primitives. The use of public-key cryptographic operations has proved to be unsuitable in embedded environments [2] due to the significant amount of processing and resources required by these operations.

The Open Mobile Alliance (OMA) [16] specifies some protocols for secure communication between mobile clients and servers at both the transport layer and the application layer. These protocols are mainly targeted towards devices that use the Wireless Application Protocol (WAP) [17]. WAP-based applications have experienced a decline in their user base lately due to a number of weaknesses in WAP, which in-device applications can overcome. WAP is a browser-based technology; this makes it unsuitable for developing applications that require fast user interaction and complex client-side logic. Although browser-based applications have shown a great success in the wired Internet, this isn’t the case in wireless environments due to the small display screens of most wireless devices and the relatively excessive network latency experienced in wireless networks. Furthermore, this browser-based type of WAP applications requires a constantly available network connection, since the applications themselves reside on the server, which results in relatively large network traffic compared to in-device applications that run entirely on the client device and perform network interactions only when necessary with the server for the sake of retrieving raw bytes of data required by the application. WAP 2.0 uses XHTML as its markup language instead of WML used in WAP 1.x. This further increases the network traffic due to the enhanced features and richer tags that XHTML possesses over WML.

SPARTA on the other hand is designed to be independent of the underlying communication protocols or wireless network infrastructure.

3. **SPARTA Design and Architecture**

This section provides an overview of the design and architecture of SPARTA on the client and the server sides. An abstract view of the major components of the security model is shown in Fig. 1.

3.1. **The Network Security Policy**

The network security policy specifies the security operations of the enterprise application needed for securing network data over the wireless links. The source information of this policy is present in the policy configuration file and it is internally represented.

---

**Fig. 1.** (a) Client-side SPARTA Architecture. (b) Server-side SPARTA Architecture.
in the runtime system by a special object known as the policy object (see Fig. 1). SPARTA's network security policy consists of two main parts used by the client and server to customize and configure their Security Engine components at policy loading time. The Security Engine is the component responsible of taking security decisions and carrying out security-related operations at runtime. The first part specifies a set of security-related attributes and parameters while the second part provides a representation of the client/server network interactions and their format. The security-related attributes specified in the first part are required by the Security Engine to control the authentication, confidentiality, integrity, and key management operations on the client and server. The attributes supported by SPARTA’s network security policy are the encryption algorithm, the hashing algorithm, the authentication mode, the session keys lifetime, and the user password lifetime.

The second part in the security policy provides a specification of the format of the network interactions and messaging between the client and server at both the request and response levels. According to this specification every network request and response belongs to one of four modes:

- A **Secure_All** mode: this mode states that all the request or response data must be secured and specifies the strength of encryption to be applied on this data. Four encryption levels are supported by SPARTA: a **High_Security** level which is equivalent to 256-bit AES encryption (By AES encryption we mean an encryption complying with the AES standard as provided by the National Institute of Standards and Technology (NIST) and we don’t specify the Rijndael algorithm itself), a **Medium_Security** level which is equivalent to 192-bit AES encryption, a **Low_Security** level which is equivalent to 128-bit AES encryption, and a **No_Security** level which represents no encryption or security on the data.

  - A **Secure_Range** mode: A request/response transaction belonging to this mode indicates that the request/response data is a message whose content is to be secured according to one of the security levels supported by SPARTA. The representation of this mode in the security policy specifies the message sections to be secured in byte ranges together with the level of security to be applied on each range. The term “byte” in this context doesn’t represent a physical unit of data storage but rather a logical unit whose specification depends on the type of data being secured by SPARTA. This logical and generic representation of the range of data to be secured is very essential since it allows the specification of these ranges without any dependency on the data encoding mechanism and representation.

  - A **Secure_Fields** mode: the **Secure_Fields** mode provides a more flexible and user-
friendly alternative to the Secure Range mode by allowing the specification of variable-length fields and data to be secured by SPARTA. This mode represents requests/responses which consist of a set of application specific fields that are usually wrapped by static data. SPARTA allows the embedding of this static data in the network security policy (this helps in reducing the network traffic) together with a specification of the position of the fields within the static data and the level of security to be applied on each field.

- A Secure_None mode: this is the opposite of the Secure_All mode and specifies that no request/response data must be secured.

3.1.1. The Network Policy Object. The Network Policy object is a compact, internal representation of the network security policy in the memory of the client and the server. This object is initialized at policy loading time and is the primary source that the Security Engine consults for taking security decisions and carrying out security-related operations at runtime.

3.1.2. The Network Policy Configuration File. To allow for the flexible configuration and customization of the network security policy by users and system administrators, the policy information is stored in an external store outside the runtime environment. The design recommends storing the external policy representation on the server-side; however, neither the exact location of the store on the server nor its type is specified by SPARTA’s design. These issues are implementation dependent: the design of the security policy does not specify nor depend on the external representation of the policy information. The implementation can flexibly choose to store the policy information in a serialized binary file, a relational database, a directory service controlled by the Lightweight Directory Access Protocol (LDAP), a flat text file, or any other type of persistent storage. The sample implementation developed in Java stores the policy information in a flat ASCII file format on the server file system. This file can be easily composed and configured using a simple text editor or a graphical tool. A sample network policy configuration file for a simple mobile banking application is shown in Fig. 2a. SPARTA supports two symmetric encryption algorithms, AES Rijndael and Twofish, and two hashing algorithms, SHA-1 and MD5. These algorithms were chosen for their speed and efficiency in wireless environments [3]. It is worth emphasizing that SPARTA in theory can generically support any number of encryption and hashing algorithms. However, adding additional algorithms is practically infeasible due to the limited storage capabilities on current wireless devices. Most current low-end wireless platforms such as J2ME do not support dynamic resource loading and thus all the algorithmic resources have to reside statically in the address space of the wireless client. Therefore, supporting a large number of encryption ciphers and hashing algorithms would increase the storage requirements of the client-side SPARTA implementation. With the technological advance in wireless devices, SPARTA will necessarily increase its algorithmic support. Two authentication modes are supported by SPARTA; these are the Challenge/Response and the Password_Digest authentication modes. We will further elaborate on these in the discussion on the Security Engine in Section 3.4. The policy file also specifies the format of the client/server network interactions. As shown in Fig. 2a, every request/response pair is identified by a unique key identifier that is used by the Security Engine to identify the particular interaction type and to secure it based on the format and mode of the request and response. The authentication process is considered a special interaction type and may be viewed as having the key identifier 0.

SPARTA automatically determines the request/response modes according to the format of the request and response.

SPARTA does not carry out a handshake at the start of the connection to decide on common encryption and hashing algorithms on the server and client sides. These algorithms should be specified explicitly in the security policy by the user or the system administrator. Avoiding such a handshake procedure is considered a very important factor which contributes to the efficient operation of SPARTA on wireless networks with high latency and low bandwidth and on wireless devices with limited resources and slow processors.

3.2. The Storage Security Policy

The Storage Security Policy specifies the mechanisms and procedures for securing the storage of application data on the client device. This policy is handled very much like the network security policy where its source information is present in the storage policy configuration file and is internally represented in the runtime system by the storage security policy object. The storage security policy consists of two main parts: the first part specifies the encryption and hashing algorithms to be used in securing the storage of local data and indicates whether the integrity of the local cache contents should be checked every time the application loads on the client device. The second part in the security policy controls the level and scope of the security operations to be applied on the locally-stored messages. According to this specification, the security of every local message to be stored belongs to one of three security modes similar to those specified in the network security policy in Section 3.1:

- A Secure_All mode: this mode states that all the contents of the local message must be secured and specifies the strength of the encryption to be applied on this message. The storage security policy supports the same encryption levels specified in the network security policy in Section 3.1.
- A Secure_Range mode: this mode specifies the local message sections to be secured in byte ranges together with the level of security to be applied on each range.
- A Secure_None mode: this mode states that the local data must be stored as is without any encryption.
Moreover, the second part of the storage security policy gives the wireless application user or administrator the ability to enforce the integrity of every local message by storing an encrypted message digest together with this specific message in the client cache. This option allows the client application to check the integrity of the stored data whenever this data is accessed or at application startup.

3.2.1. The Storage Policy Object. The storage policy object is a compact, internal representation of the storage security policy in the memory of the client and server. This object is initialized at policy loading time.

3.2.2. The Storage Policy Configuration File. The storage policy configuration file is similar in structure and format to the network policy configuration file. A sample storage policy configuration file is shown in Fig. 2b. The first 2 entries in this file specify the encryption and hash algorithms to be used in securing the confidentiality and integrity of locally-stored data. SPARTA uses the same cipher suite presented in Section 3.1 for securing local data. The third entry is represented by the Integrity_Check_At_Startup parameter. If this parameter is enabled, then the integrity of every stored field whose Integrity_Enforcement attribute is set to Yes, is checked at application startup. This initial integrity check is very important for avoiding modification attacks on the client local cache.

Every stored field is identified in the storage security policy by a unique key identifier known as the storage field Id. For each stored field entry, the storage security policy defines 2 parameters, the Encryption parameter and the Integrity_Enforcement parameter. The Encryption parameter specifies the level and scope of the encryption operations based on one of the security modes presented above. SPARTA automatically determines the particular security mode according to the format of the Encryption parameter value. The Integrity_Enforcement parameter specifies whether SPARTA should store an encrypted message digest with the local field to verify its integrity at application startup (if the Integrity_Check_At_Startup parameter is enabled) or whenever this field is accessed by the SPARTA API.

3.3. The Policy Loader

In this section the term “policy” refers to both the network and storage policies unless otherwise specified. In SPARTA, the Policy Loader component is responsible of carrying out the policy loading operation from its central location on the server to the client. In the next sections we present several design issues that were considered when designing the Policy Loader.

3.3.1. Policy loading time. The policy loading time is the time at which the client requests loading the security policy to initialize the Policy object at runtime. SPARTA loads the security policy at the time it is first needed, that is at authentication time (lazy loading).

3.3.2. Policy compaction. The policy information present in the policy configuration file was designed to be human-readable. Transferring this information as-is over the network to the client device will not only produce unnecessary increase in the network traffic, but will also increase the storage requirements due to the policy caching mechanism (see Section 3.3.4), and the processing load on the client device due to the large number of string operations necessary to parse the policy information and to identify the request and response abstractions. For this reason, SPARTA introduces the Policy Compactor component responsible for parsing the policy configuration file and converting it into a compact binary form before transmitting it to the client.

3.3.3. Policy security. The security policy in SPARTA contains sensitive information that controls the various security operations in a wireless enterprise application. Any malicious modification to this information may lead to dangerous security attacks (consider the scenario where all the request/response modes are maliciously modified to the Secure_None mode or where all the security levels are changed to the No_Security level). We used MACs (symmetrically-encrypted message digests) to protect the integrity of the policy information when loaded over the network. We assume that the client and server securely share a 64-bit password and a 64-bit shared secret. In SPARTA the agreement on these parameters takes place at service registration time. The policy integrity assurance is illustrated in Fig. 3. It should be noted here that SPARTA considers the protection of the policy
configuration file on the server's file system against unauthorized modifications to be the responsibility of the server's underlying operating system.

3.3.4. Policy caching. Policy caching on the client-side improves the policy loading time by using a local copy of the security policy instead of loading it remotely from the server every time it is requested. However, due to the possible modification of the policy configuration by users and administrators, good care should be taken to ensure that the policy information is kept up to date in the local cache. The process of detecting the modification of the policy configuration is implementation dependent. In the sample implementation, we used message digests to check for possible updates in the policy configuration. The process is shown in Fig. 3.

3.4. The Security Engine

The Security Engine component is responsible for providing authentication and data confidentiality and integrity security services. It participates in carrying out an efficient and secure key management mechanism and ensures the content-based storage security of the sensitive parameters on the client-side. The security services supported by the Security Engine are controlled and configured based on information present in the Network and Storage Policy objects. The Security Engine is the only component that can be accessed by the application. The interaction between the Security Engine and the application is done through the SPARTA API that hides the complexity of security programming and operations from the application developer. The main services supported by the Security Engine are discussed in the following sections.

3.4.1. Key Management. SPARTA implements a flexible and secure session-key management mechanism that suits a wide range of wireless enterprise systems with a variety of performance and security requirements. In this mechanism, the session keys used for data encryption and decryption are generated randomly on the server side. The lifetime of these keys is specified in the Session_Keys_Life attribute in the security policy. It should be noted that the lifetime of the session keys has a major impact on the security of the system and its performance. As the lifetime decreases, the security increases since the network data in this case will be encrypted by frequently changing keys which reduces the risk that an attacker will gather useful information about the encrypted network data, and makes successful analysis less probable. However, this increase in security comes at the expense of performance. For this reason, the inclusion of the Session_Keys_life attribute in the security policy gives enterprise users a great deal of flexibility in setting this attribute according to the security requirements of their application environment and the capabilities of their devices. The session-key management mechanism uses three keys which represent the security levels supported by SPARTA. The first key is a 128-bit key, the second key is a 192-bit key, and the third key is a 256-bit key. These keys are used by the Security Engine to provide the Rijndael/Twofish low, medium, and high security levels respectively. The key management mechanism is illustrated in Fig. 4.

Another important issue that must be addressed is securing the storage of the shared secret on the client and server. To achieve this security on the server side, SPARTA uses the database server access control and encryption security mechanisms. On the client-side, the shared secret is encrypted using the client’s password (64-bits) padded with itself and stored encrypted in the client's local cache.

3.4.2. Authentication. SPARTA supports two authentication modes: the Challenge/Response mode and the Password_Digest mode. Specifying the required mode is done externally by setting the Authentication_Mode attribute in the network policy configuration file without any modification to the application. The two supported authentication modes depend on a 128-bit random number, the challenge, that the Policy Loader requests from the Initializer component at policy loading time. In the Challenge/Response mode, the client-side Security Engine appends the password to the random challenge and encrypts the concatenation using the key that corresponds to the security level specified in the
request mode of the Authentication interaction type in the security policy. The Security Engine will then append the username to the encrypted result and send the combination to the server. The Security Engine of the corresponding user on the server will use the username to lookup the password in the server's persistent storage. The retrieved password will be appended to the random challenge that was previously produced by the server, and the result is encrypted with the same security level specified in the request mode of the Authentication interaction type in the network security policy. The encrypted value is compared to the received one; if the two values are equal, the client is authenticated; otherwise, the client access will be denied.

In the Password_Digest authentication mode, the client does not send the password to the server, but it sends a message digest of the password instead. The password digest value will be appended to the random challenge that was previously produced by the server, and the result is encrypted with the same security level specified in the request mode of the Authentication interaction type in the network security policy. The Security Engine will then append the username to the encrypted result and send the combination to the server. The Security Engine of the corresponding user on the server will use the username to lookup the password in the server's persistent storage. The retrieved password will be digested and appended to the random challenge that was previously produced by the server, and the result is encrypted with the same security level specified in the request mode of the Authentication interaction type in the security policy. The encrypted value is compared to the received one; if the two values are equal, the client is authenticated; otherwise, the client access will be denied.

The main difference between the Password_Digest and the Challenge/Response authentication modes is that the former does not send the user password on the wireless links but rather a hash of this password. For this reason, the Password_Digest authentication mode is considered relatively more secure than the Challenge/Response authentication mode. From a performance point of view, the Challenge/Response authentication mode avoids two hashing operations, one on the client-side and the other on the server side. Although these hashing operations do not impose a great performance overhead, they may have a remarkable effect on the authentication time on low-end wireless devices with limited resources and processing power.

Usually a wireless device is not shared among multiple users. In SPARTA, only the user who owns the wireless device and knows the application password can decrypt the shared secret and, as a result, the ciphering keys to be authenticated to the system. This, in addition to the strong and high-entropy password generation, helps in preventing password guessing and dictionary attacks. Moreover, the 128-bit random challenge, gives a blend of randomness to the client’s authentication data so that an eavesdropper examining this data over the wireless network will get different values every time the client sends his authentication data to the server. This will also prevent the intruder...
from making use of previous session authentication data to gain access to the system. In case the wireless device is lost, stolen, or even sold, SPARTA prevents any possible dictionary or brute force attacks on the password by locking the user account on the server after a specific number (default 5) of successive failed authentication attempts.

3.4.3. Data confidentiality and integrity. To support a flexible encryption scheme, SPARTA specifies the format of the network interactions and messaging in the network security policy and provides a standard interface through which the application and the Security Engine communicate and interact. Through this standard interface, the Security Engine abstracts the security operations and hides their complexity from the application. The application developer only needs to supply the Security Engine with a representation of the data to be secured and the interaction type identification number that specifies how this data is to be secured and to what degree. It is the responsibility of the client-side Security Engine to perform the encryption and hashing operations of the application data and to construct the secure request based on the request mode and format in the security policy. The Security Engine on the server-side accepts the secure request, performs the necessary integrity verification and decryption operations on this request, and constructs the original request message as sent by the client application and delivers it to the server-side application. This is done based on the request mode in the network security policy.

Sending a secure response from the server to the client follows the same procedure. In this case the client-side and the server-side Security Engines will switch roles and the security operations and their strengths will be based on the response mode of the particular interaction type in the network security policy. Securing the network data confidentiality and integrity is illustrated in Fig. 5. It should be noted that the algorithm used by the Security Engine in constructing and delivering the secure requests and responses depends on the request/response modes discussed in Section 3.1.

The policy-based security of the client-side stored data is also provided by the Security Engine component. For storing a particular data field, the client-side application supplies the Security Engine with this data field together with the storage type identifier. The Security Engine uses this identifier to find the specific storage entry in the storage security policy and to provide the encryption and integrity security services on the data field. Securely retrieving stored information from the client local cache follows a similar procedure. The application supplies the Security Engine with the storage type Id which is used by the later to perform the decryption and integrity verification operations on the stored data field based on the information present in the storage security policy. This is illustrated in Fig. 6.

The ciphering mode used in SPARTA’s implementation is the cipher block chaining (CBC) mode. The padding scheme used abides with the PKCS #7 [14] standard.

4. SPARTA Implementation

SPARTA was designed in a platform-neutral manner and the design features presented in Section 3 can be implemented on available wireless client platforms and enterprise backend systems. A sample implementation for SPARTA was developed for the J2ME platform on the client-side and the J2EE platform on the server-side. In general to implement SPARTA on a specific wireless platform, this platform must satisfy the following main requirements:

1. The wireless platform must be portable on a wide range of devices.
2. It must be able to perform wireless network interactions to support the communication between a mobile device and an application server. The wireless application must be capable of interfacing with enterprise systems, databases, corporate intranets, and the Internet.
3. It must support the cryptographic primitives for providing the confidentiality, integrity, and authentication security services such as encryption, hashing, and MAC algorithms.
4. It must give the mobile application the ability to store persistent local data on the wireless device.
5. It must provide some basic string parsing and processing methods for manipulating character-based data.

The J2ME platform satisfies all the above requirements.

1. J2ME is a portable platform based on Java technology.
2. J2ME supports an elegant networking model suitable for enterprise wireless application development. It provides the javax.microedition.io CLDC package which contains classes for input/output including networking I/O. Furthermore, the MIDP on top of the CLDC provides the HttpURLConnection interface which devises the necessary methods needed for encapsulating an HTTP connection.
3. J2ME doesn’t directly support cryptographic algorithms, however numerous third-party cryptographic packages and toolkits are available and developed to be compatible with the J2ME wireless platform. SPARTA’s sample implementation presented in this Section uses in some of its cryptographic algorithms implementations provided by the Bouncy Castle cryptographic package [18].
4. J2ME supports a Record Management System (RMS) API that gives MIDP applications local, on-device data persistence services. At the API level, a record store is represented by an instance of the javax.microedition.rms.RecordStore class. All RMS classes and interfaces are defined in the javax.microedition.rms package.
J2ME provides the String and StringBuffer classes that support a useful bunch of string parsing and manipulation methods.

In this section we present a brief overview of this SPARTA implementation, including an overview of the server-side environment, and the client-side environment that SPARTA was implemented on.

### 4.1. The server-side environment
SPARTA server-side components were implemented in accordance with the J2EE specifications [6]. The Sun J2EE reference implementation server version 1.3.1 was used for running the server-side implementation. The persistent storage services were provided by the Cloudscape database server, a pure Java implementation by IBM that provides small footprint DBMS.

### 4.2. The client-side environment
SPARTA's client-side implementation complies with the CLDC/Mobile Information Device Profile (MIDP) 1.0 and 2.0 specifications [5]. Choosing J2ME/CLDC platform for implementing the client-side SPARTA

---

**Table 1. Encryption/Decryption Performance**

<table>
<thead>
<tr>
<th>Kbit/sec</th>
<th>Rijndael Encryption/Decryption</th>
<th>Two fish Encryption/Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128 bits</td>
<td>192 bits</td>
</tr>
<tr>
<td>Sony Ericsson P800</td>
<td>348.59</td>
<td>300.62</td>
</tr>
<tr>
<td>Nokia 6600</td>
<td>182.04 / 110.32</td>
<td>156.78 / 104.02</td>
</tr>
</tbody>
</table>

**Table 2. Hashing Performance**

<table>
<thead>
<tr>
<th>Kbit/sec</th>
<th>MD5</th>
<th>SHA-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sony Ericsson P800</td>
<td>1057.03</td>
<td>697.19</td>
</tr>
<tr>
<td>Nokia 6600</td>
<td>541.62</td>
<td>315.07</td>
</tr>
</tbody>
</table>

**Table 3. Time Overhead Per Stored Byte on Sony Ericsson P800**

<table>
<thead>
<tr>
<th>× 10^-6 sec</th>
<th>Rijndael Encryption/Decryption</th>
<th>Two fish Encryption/Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128 bits</td>
<td>192 bits</td>
</tr>
<tr>
<td>MD5</td>
<td>30.5 / 45.65</td>
<td>34.17 / 48.1</td>
</tr>
<tr>
<td>SHA-1</td>
<td>34.41 / 49.60</td>
<td>38.10 / 52</td>
</tr>
</tbody>
</table>

**Table 4. Time Overhead Per Stored Byte on Nokia 6600**

<table>
<thead>
<tr>
<th>× 10^-6 sec</th>
<th>Rijndael Encryption/Decryption</th>
<th>Two fish Encryption/Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128 bits</td>
<td>192 bits</td>
</tr>
<tr>
<td>MD5</td>
<td>58.71 / 87.28</td>
<td>65.80 / 91.67</td>
</tr>
<tr>
<td>SHA-1</td>
<td>69.33 / 97.90</td>
<td>76.41 / 102.29</td>
</tr>
</tbody>
</table>

**Table 5. RMS Access Performance**

<table>
<thead>
<tr>
<th>Kbit/sec</th>
<th>Read</th>
<th>Write</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sony Ericsson P800</td>
<td>9362.28</td>
<td>704.69</td>
<td>4096</td>
</tr>
<tr>
<td>Nokia 6600</td>
<td>4369.06</td>
<td>383.25</td>
<td>2114.06</td>
</tr>
</tbody>
</table>
was done on purpose to prove the effectiveness and efficiency of SPARTA on low-end platforms with limited features and capabilities. The client-side implementation was developed using the Sun Wireless Toolkit version 2.2 and tested on the Sony Ericsson P800 and the Nokia 6600 J2ME-enabled phones.

5. Performance Analysis

The performance of the SPARTA project was tested in a real wireless environment by deploying the client-side implementation package on the Sony Ericsson P800, and the Nokia 6600 J2ME-enabled phones.

The server-side implementation package was deployed on the J2EE reference server v1.3.1 from Sun Microsystems running under Microsoft Windows 2000. A simple mobile banking application, which uses SPARTA security services, was installed on the client and the server together with some utility modules for measuring the performance of the bulk encryption and hashing operations on the Java phones. The networking interactions between the J2ME client and the J2EE server were performed using HTTP/TCP running over a General Packet Radio Service (GPRS) connection.

5.1 Encryption/decryption and hashing performance

The encryption, decryption and hashing performance, using 4KB of data, on Sony Ericsson P800 and Nokia 6600 is presented in Table 1 and Table 2 respectively.

5.2. Memory Requirements

Controlling the size of the sample client was a major design goal from the start without jeopardizing the clean interface between the various components comprising the architecture and its ease of use. The size of SPARTA client code is approximately 34 KB. This is very acceptable, and if SPARTA is to be added to the base J2ME implementation, it would only increase its size by about 10%.

5.3. Network-Related Measurements

The average Network-related measurements, recorded when testing the mobile banking application are as follows: the authentication operation including policy loading took 5.01 sec; the policy loading operation from the server took 2.81 sec.

5.4. Storage-Related Measurements

Table 3 and Table 4 present the time overhead per byte for applying SPARTA’s storage security mechanisms (encryption/decryption and hashing) on locally-stored messages. These values are derived from the results of Table 1 and Table 2. The storage overhead per stored message depends on the hashing algorithm used for data integrity enforcement. This overhead is 20 bytes when SHA-1 is used and 16 bytes when MD5 is used.

MIDP stores local application data in non-volatile memory, using the RMS storage system. The average RMS access time for reading, writing, and deleting data from the RMS record store is presented in Table 5.

The average time to perform a user transaction (including the storage security services) was found to be 2.49 sec.

The above results show that the mobile banking application using SPARTA has operated efficiently on a real wireless network. The operations requiring a network connection between the client and the server are considered very responsive. For the numbers shown above, most of the operation time is spent as network roundtrip delay, which is a factor that we cannot control since it depends on the wireless network infrastructure. The average GPRS network roundtrip time was found to be approximately 1.95 seconds.

6. Conclusion

In this paper we presented SPARTA, a new policy-driven security architecture for wireless enterprise applications. We provided an overview of SPARTA’s design and architecture, and showed a sample implementation using the J2ME/J2EE platforms; excellent results were obtained for the performance of this implementation when tested in a real wireless environment.

7. References


