A Security-by-Contracts Architecture for Pervasive Services*

N. Dragoni F. Massacci
University of Trento
surname@dit.unitn.it

C. Schaefer T. Walter
DoCoMo Euro-Labs
surname@docomolab-euro.com

E. Vetillard
Trusted Logic
eric@trusted-labs.fr

Abstract

Future pervasive environments will by characterised by pervasive client downloads*: new (untrusted) clients will be dynamically downloaded in order to exploit the computational power of the nomadic devices to make a better use of the services available in the environment.*

To address the challenges of this paradigm we propose the notion of security-by-contract *(*S×C*), as in programming-by-contract, based on the notion of a mobile contract that a pervasive download carries with itself. It describes the relevant security features of the application and the relevant security interactions with its nomadic host.*

In this paper we describe the layered security architecture of the S×C *paradigm for pervasive security, the threats and mitigation strategies of security services and sketch some interaction modalities of the security services layer.*

1 Introduction

The paradigm of pervasive services [1] envisions a nomadic user traversing a variety of environments and seamlessly receiving services from other devices. Yet, the challenge is broader than this "simple" distributed service vision because it does not consider the possibilities that open up when we realize that the smart phone in our pocket has *already* more computing power than our 10 years old PC.

Current pervasive services, including context-aware services, hardly exploit this computational power. Information is provided to the mobile user based on his location but the computing infrastructure is usually centralised [7]. Even when it is decentralised to increase scalability it does not exploit the devices' power (e.g. [3, 2]).

There is a new challenge ahead: *pervasive client downloads*: when traversing environments the nomadic user does not only invoke services in a web-service-like fashion (push or pull mode) but also download *new* applications that can exploit the computational power device to make a better use of the (new) services available in the environment.

A tourist landing in a historical city might download at the airport a *tourist guide* application that can route her rented car to her favourite touristic hotspots. The application is configured with mentioned touristic hotspots and, in order to determine the route to those hotspots, the application needs to interact with the car's navigation system to determine the current location and to update the route planning (but only if confirmed by the driver), and might send travel tips to selected driver's companions. This is a pervasive service download because it offers local services, we want it exactly where we need it (e.g. via a bluetooth link at the airport), without bothering a long and frustrating web search before departing. We use it on our mobile or PDA without connecting to a remote route planner each time.

This scenario violates the model of software download of the Java [6] and .NET security architecture [16, 10]:

- A pervasive download is untrusted code whose security properties we cannot check and whose code signature (if any) will not bring any degree of trust;
- it should be sandboxed, its interaction with the environment and the devices own data should be limited;
- yet this is against the business logic, as we made this pervasive download precisely to have lots of interaction with the pervasive environment!

The Contribution of this Paper. In this paper we advocate the notion of *Security-by-Contract (*S×C*)* as a solution to the pervasive download problem. Loosely speaking, an application should come with a *contract* containing a description of the relevant features of the application and the relevant interactions with its host platform. A mobile platform could specify its security requirements, a *policy*, which should be matched by the application's contract.

The contributions of the paper are threefold: (1) we describe the layered security architecture of the S×C paradigm for pervasive security, (2) we discuss the threats and mitigation strategies for the security services and (3) sketch some interaction modalities of the security services layer.

*This work is partly supported by the project EU-IST-STREP-S3MS.

We define the security architecture as in RFC-2828: *[the] set of principles that describe (a) the security services that a system is required to provide to meet the needs of its users, (b) the system elements required to implement the services, and (c) the performance levels required in the elements to deal with the threat environment.* In this paper we focus on the first two aspects which leads to a functional design of the security architecture. The performance issue will be addressed during its implementation.

In the next section (§2) we present a prototypical example of pervasive client downloads, mobile games. Then we discuss the key intuition behind security by contract (§3) and the layered security architecture supporting it (§4). Security threats are analysed (§5) and mitigating services are proposed (§6). We end the paper describing some scenarios (§7) and concluding.

2 The Ancestors of Pervasive Downloads

As we said, current pervasive services are essentially local invocation of centralised services with little or no dynamic content with one notable exception: Mobile Games. On-line games are more complex than one can expect [9] and *Massive Multi-player Online Role Playing Game* (MMORPG), introduces additional challenges. Essentially, an MMORPG is a persistent, graphical, online environment which allows many users to play simultaneously [17]. Once again, most commercial games are based on multi-billion massive servers storing data and characters.

Still, new architectures of MMORPG are emerging based on peer-to-peer (P2P) technologies and a pervasive notions of gaming [13, 8]. The basic idea is that servers must be as thin as possible, essentially performing registration of the players and providing the data of the different Levels and the Rooms of the World. The players will do the rest, by interacting with other players located in the same "virtual" environment and by communicating directly with them. Most of the interactions require a P2P connection and most implementations are structured in order to ease such local interaction, using whatever communication facility is available to the players (SMS, MMS, GPRS, WiFi, Bluetooth) as shown in Fig. 1.

This paradigm reflects perfectly the business logic and the security issues of pervasive downloads:

1. Most game developers are SMEs which cannot afford the costs necessary to obtain a mobile operator's certification and thus will not run as trusted code;
2. downloaded clients require to perform a significant amount of computation and communication with other service rounds (players or other services);
3. such actions use and consume costly resources such as battery, bandwidth or time or sensitive resources such as the user address book or other security material used to authenticate it towards other players.

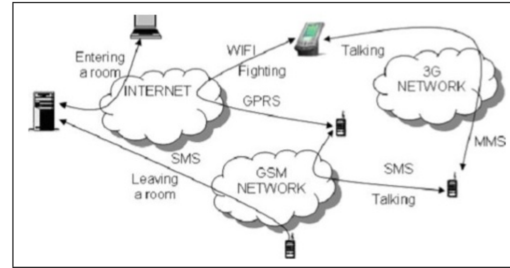

Figure 1. P2P Interaction in MMORPGs

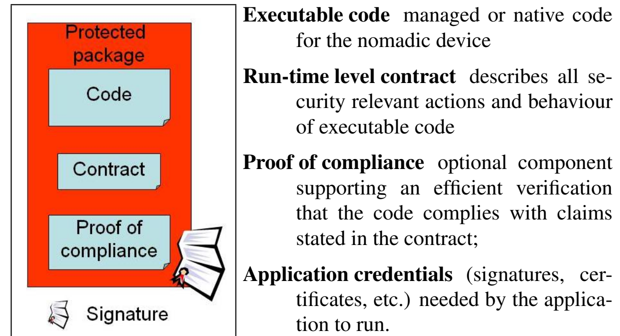

Figure 2. Components of a S3MS Application

3 Security-By-Contract Framework

The framework of S×C is shaped by four stake-holders: mobile operator, service provider or developer, mobile user and third party security service provides. Application developers are responsible to provide a description (called contract) of the security behaviour that their code exhibits.

With the security-by-contract paradigm each "application" may consist of the four components described in Fig. 2. By signing the code the developer binds the code with the stated claims on its security-relevant behaviour thus providing a semantics to digital signatures.

Definition 3.1 (Contract) *A contract is a formal, complete and correct specification of the behaviour of an application for what concerns relevant security actions (Virtual Machine (VM) API Calls, Operating System Calls).*

Users and mobile phone operators are interested in that any software deployed on their platform is secure. In other words they must declare their security policy:

Definition 3.2 (Policy) *A policy is a formal, complete and correct specification of the acceptable behaviour of appli-*

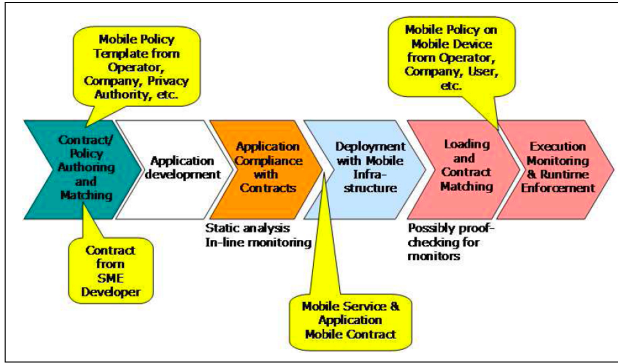

Figure 3. Application/Service Life-Cycle

cations to be executed on the platform for what concerns relevant security actions (VM API Calls, etc...).

A contract should be negotiated and enforced during development, at time of delivery and loading, and during execution of the application code by the mobile platform. Fig. 3 summarises the phases of the S×C life-cycle.

The first step is the *contract authoring* stage in which a contract for the mobile application is specified (as requirements to the application development teams or derivation from analysis of the existing application) and a policy template for the platform is provided (as defined by the template of the operator, company, etc...).

Once identified the contract that the application actually provides and the policy that the platform would like, the step of *contract matching* is performed, i.e. matching the compliance of the application as given by the contract with the requirements as defined in the policy.

We also have the problem of *compliance of the application with the contract*. In order to guarantee that an application complies with its desired contract or the policy requested on a particular platform we should consider the stage where such enforcement can be done. Enforcing at development time can be achieved by appropriate design rules. Enforcement before or at deployment can be carried out through (automatic) verification techniques. Such verifications can take place before downloading (*static verification* [11] developers and operators followed by a contract coming with a *trusted signature*) or as a combination of pre and post-loading operations (e.g., through *proof carrying code* [15] and *in-line monitors* [4, 12]); run-time enforcement can be implemented by *run-time checking* [5, 12].

All methods have different technical and business properties. Table 1 shows strengths ($\surd$) and limitations ($\times$) of each technology w.r.t. the different requirements. For instance, working on existing devices would rule out run-time enforcement and favour static analysis, code signing and signature verification on the mobile platform. Monitors may be used (for properties that could not be proved), but on-device proof would then not be possible.

Table 1. Tech. Strengths and Weaknesses

Criteria	Static Analysis	Monitors	Runtime
Works with existing devices	$\surd$	?	$\times$
Works with existing applications	?	$\times$	$\surd$
Does not modify applications	$\surd$	$\times$	$\surd$
Offline proof of correctness	$\surd$	$\surd$	$\times$
Load-time proof of correctness	$\times$	$\surd$	$\times$
May depend on run-time data	$\times$	$\surd$	$\surd$
Does not affect runtime performance	$\surd$	?	$\times$

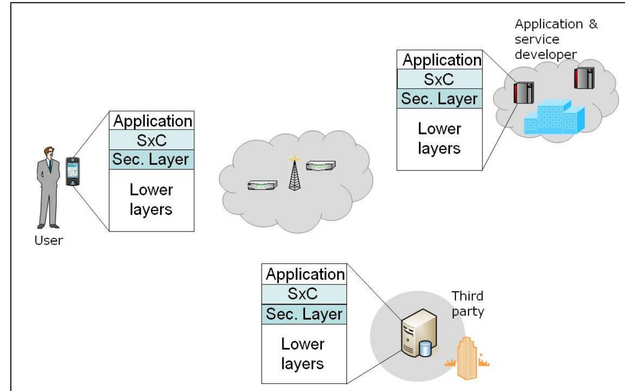

Figure 4. S×C **Architecture**

4 Layered S×C Architecture

The security architecture has the following goals:

- Supporting the application and service life cycle shown in Fig. 3 which means guaranteeing the security of the channel between parties as well as authenticity of the parties and non-repudiation of communication actions for charging and billing.
- Enabling trust relationships between stakeholders, i.e. authenticity and integrity of exchanged data elements such as code, contract and policy. Specifically, this authenticity and integrity of data elements becomes important when one thinks about that in-lining of code for contract compliance is provided by a third party.

The S×C architecture is composed of 3 layers (Fig. 4).

Application layer: where development tools run on the developer's system, and where administration and certification services run on the third party's systems. The application layer accesses services of the S×C layer.

Table 2. S×C services

Service	Self-Service	Value-Added Serv.
get(Code, Contract)	on-line	on-line
analyse(Code, Contract)	N/A	off-line
inline(Code, Contract)	N/A	off-line
inline(Code, Policy)	on-device	on-line/off-line
match(Contract, Policy)	on-device	on-line
monitor(Code, Policy)	on-device	N/A
check(Code, Contract, Proof)	on-device	on-line
proofGen(Code, Contract)	N/A	off-line
manage(Policy)	on-device	on-line/on device

S×C layer: where services enable the user to download code (from the developer or a third party) and to initiate specific S×C services such as performing in-lining of code for code and contract compliance.

A service is used *on-device* if it is invoked and executed on the very device that the user is using to run also the application. It might be executed *on-line* if a communication with the pervasive environment must be established in order to obtain the services. A service might be also offered *off-line* followed by traditional signature verification.

Not all services can be equally well used at different levels. Monitoring the code for compliance with the user-defined policies, for instance, makes sense only on the mobile device itself. On the other hand, some services can very well be executed in advance. The in-lining of code for contract compliance, for instance, can be done by a third party in advance of deploying the code on a mobile device.

In Table 2 we consider two interesting cases from the business models point of view of S×C. In the *self-service* model the device is powerful enough to perform most tasks by itself or rely on the developer to check a number of them. In the *value-added services* the S×C services are offered by a third party such as the mobile operator of the device.

Security layer: whenever S×C services require the cooperation of an external party, e.g., code and contract that is being downloaded from the developer, the S×C layer calls the services of the security layer to protect the communication between the parties. These services and the respective service primitives are further detailed in Section 6.

The interactions between the described layers are as follows (see Section 7): Assuming that the user of the mobile device is performing a download of an application from the developer then the user calls the respective method of the S×C layer (i.e. get (Code) from (Developer)). This call is mapped into a respective call to the security layer which maps it into the sequence of method calls to set up a connection with the developer, performs authentication of the developer, gets the package containing code, contract and signature, checks the signature for correctness, extracts code and contract from the received package and hands code and contract back to the calling S×C layer which in turn gives the code and contract back to the application layer.

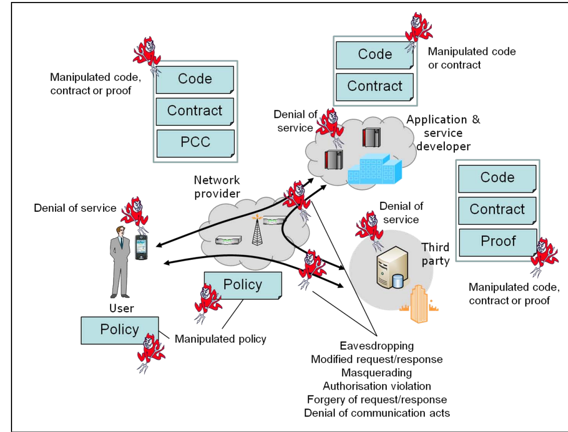

Figure 5. Threats Analysis

5 Threat Analysis

In this section, we describe possible threats and the elements these threats have an impact on, i.e. data items, communication links, processes or devices. For the threat model, Fig. 5 combines parties and data elements and identifies possible threats which we analyse below.

Threats against data elements code, contract, proof of compliance and policies. Code and contract are maintained in the domain of the application and service developer. The proof of compliance may be given as PCC (proof-carrying-code) in which case it is maintained in the domain of the developer. Policies are a concern of the user and the mobile network operator. These components are bound into packages (Fig. 2). A possible threat is the manipulation of code, contract, proof and policy.

Threats against communication links. Communication between the stakeholders is as follows:

1. the user may get code, contract and optionally proof from the developer;
2. user involves a third party to perform code/contract matching, proof verification, contract/policy matching;
3. developers may involve a third party to attest code and contract compliance.

The following major threats are considered: Eavesdropping, Modifying request or response, Masquerading, Forgery of request or response, Authorisation violation. The first four

threats obviously impact the overall security of S×C as no guarantees on the authenticity and integrity of S×C components and policy can be given and, thus, execution of code may harm the user's device. Not doing proper authentication and authorisation may give parties access to code they are not entitled to access or, on the other hand, code access may be charged to parties that have not accessed said code.

Threats against parties. DoS attacks can be run against any of the parties and may block them from providing services. E.g. in order to run downloaded code the S×C operation for contract and policy matching (Table 2) has to be executed. If this can not be done by the user on his mobile device, then this operation has to be externally executed by a third party. Running a DoS attack against the third party makes the service unavailable. Consequently, according to the S×C paradigm the user can not be execute the code. Similar scenarios apply for the other parties as well (code cannot be accessed from the developer or results of the matching cannot be given back to the user).

In order to mitigate those threats a number of security services must be put in place.

6 Security Services and Primitives

The basic component of the architecture is the *nomadic device* which requires the following sub-components:

- Key store for asymmetric and symmetric keys. The key store should be protected by a passphrase, biometric trait, etc. in order to control access.
- Certificate store for own and third party certificates. Among the third party certificates should be all root certificates for parties participating in off-line services.
- Cryptographic software implementing the required cryptographic algorithms for en- and decryption, hashing and signing as well as signature verification.

The detailed data processing and communication actions depend on the various options (see Table 2), but some secure communication services are necessary:

- For communication between user and application provider it is required that the latter authenticates with the former and, if charging and billing is an issue because the download is not for free, also vice versa.

  As part of the authentication process, a protected channel can be established so that exchanged data is protected for confidentiality to avoid disclosing the data against unauthorised third parties.

  In order to achieve integrity, the code and contract should be hashed and the hash be transmitted to the user for verification against the received data.

  If, however, communication does not run over a protected channel then code and contract should be signed by the developer so that the user can identify the origin and correctness of received data.
- For the communication between user and third party, the same considerations as presented before apply. If the user communicates its platform policy to the third party then this should be done over a secret channel.
- The same arguments apply for the communication between third party and developer, i.e. if the third party deals with the downloading of code and contract and policy matching on behalf of the user.

If the download of code is being charged and billed then accounting of the download is to be performed and non-repudiation of the code download is required. This can be extended in order to cover the situation that the contract and policy matching is charged and billed by the third party.

7 Interaction Patterns

In this section we discuss some scenarios that represent different business cases for value-added service offerings.

Policy and contract as well as policy matching on mobile device. The process of downloading code is initiated by the user. The user has to authenticate with the server which holds the requested code. If required the server authenticates with the user's device, thus we authenticate mutually. After authentication has taken place, the code is downloaded onto the user's mobile device. Subsequent to the download, the S×C layer of the proposed security architecture (Fig. 4) is taking control of the further steps until execution). Assuming that the downloaded code arrives as in Fig. 2, the following steps are to be done:

1. The public key of the signing authority has to be retrieved and the signature verified.
2. The contract has to be extracted from the packet.
3. The platform policy has to be retrieved.
4. Contract and platform policy are forwarded to the contract-policy matching module (i.e. operation match(Contract, Policy) is called).
5. If matching is successful the code is loaded and executed; eventually execution is done under the control of a monitor (i.e. monitor(Code, Policy) is called).

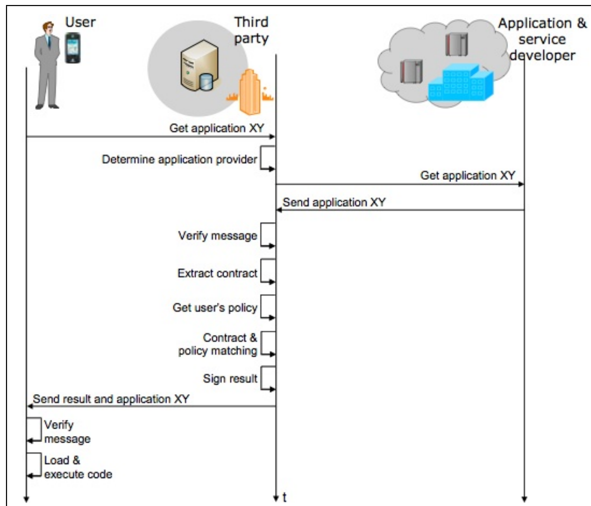

Figure 6. Download and service via portal

Policy on mobile device and contract and policy matching by third party. With this scenario we build on the previous scenario, i.e. the code is downloaded onto the user's mobile devices, but extend the scenario. A third party is involved that performs the contract and policy matching. So, the user's device sends the contract and policy to the third party. In this scenario, we assume that the user's device sends a signed message including contract and policy to the third party. Upon reception of the message the third party verifies the message's signature, extracts contract and policy, performs contract-policy matching by calling operation match(Contract, Policy) of the S×C layer, and subsequently returns the signed results back to the user's device.

Policy with third party and contract and policy matching done by third party. The last option being considered is the one where the third party functions as a ”portal”, i.e. the only interface of the user from his mobile device to the environment is via this portal. The interactions of the user with the portal are shown in Fig. 6.

8 Conclusion

In this paper we have proposed the notion of *Security-by-Contract (*S×C*)*, as in programming-by-contract, based on the notion of a mobile contract that a pervasive download carries with itself. A contract describes the relevant security features of the application and the relevant security interactions with its mobile host.

The main contributions of the paper can be summarized as follows. First we have described the layered security architecture of the S×C paradigm for pervasive security. Then we have discussed the threats and mitigation strategies for security services. Finally, we have sketched some interaction modalities of the security services layer.

The current activity is towards implementing the architecture in a realistic simulation environment such as the MOAP platform used by DoCoMo [14].

A challenge problem still open concerns the *dynamic negotiation* of contracts: the code provider should be able to negotiate with the host platform a specific contract, i.e. the two agents should reach an agreement on the set of relevant security actions that should match. Moreover, such actions can be interdependent with each other: the choice of which rules to negotiate could depend on what choice has been previously made by an agent for other rules.

References

[1] J. Bacon. Toward Pervasive Computing. *IEEE Perv.*, 1(2):84–86, 2002.
[2] D. Chakraborty, K. Dasgupta, S. Mittal, A. Misra, A. Gupta, E. Newmark, and C. Oberle. Businessfinder: harnessing presence to enable live yellow pages for small, medium and micro mobile businesses. *IEEE Comm.*, 45(1):144–151, Jan. 2007.
[3] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet. *IEEE Network*, 13(4):6–15, 1999.
[4] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. Technical report 2003-1916, Department of Computer Science, Cornell University, 2003.
[5] U. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proc. of IEEE SS&P'00*, Washington, DC, USA, 2000.
[6] L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.
[7] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. *WiNet*, 8(2 - 3):187–197, 2002.
[8] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *Proc. of IEEE Infocom*, 2004.
[9] D. Kushner. Online gaming demands heavyweight data centers. *IEEE Spectrum*, 42(7):34–39, July 2005.
[10] B. LaMacchia and S. Lange. *.NET Framework security*. Addison Wesley, 2002.
[11] X. Leroy. Bytecode verification on java smart cards. *Softw. Pract. Exper.*, 32(4):319–340, 2002.
[12] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *IJIS*, 4(1–2):2–16, Feb. 2005.
[13] M. Merabti and A. E. Rhalibi. Peer-to-peer architecture and protocol for a massively multiplayer online game. In *Proc. of GLOBECOM'04*, pages 519–528, Dallas, TX, 2004, 29 Nov.-3 Dec. 2004. IEEE.
[14] H. Tsuji, K. Ohno and T. Saito. “MOAP”, Software Platform for FOMA Terminals. *NTT DoCoMo Technical Journal*, 1(1), June 2005.
[15] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, pages 61–91. Springer-Verlag, London, UK, 1998.
[16] N. Paul and D. Evans. .NET Security: Lessons Learned and Missed from Java. In *Proc. of ACSAC'04*, 2004.
[17] N. Yee. The Psychology of MMORPGs: Emotional Investment, Motivations, Relationship Formation, and Problematic Usage. In R. Schroeder and A. Axelsson, editors, *Avatars at Work and Play: Collaboration and Interaction in Shared Virtual Environments*, London, 2005. Springer-Verlag.