Security-By-Contract for the Future Internet *

Fabio Massacci[1], Frank Piessens[2], and Ida Siahaan[1]

[1] Universita' di Trento, Italy name.surname@disi.unitn.it
[2] Katholieke Universiteit Leuven, Belgium name.surname@cs.kuleuven.be

1 The Future Internet

With the advent of the next generation java servlet on the smartcard, the Future Internet will be composed by web servers and clients silently yet busily running on high end smart cards in our phones and our wallets. Thus we can no longer accept the current security model where programs can be downloaded on our machines just because they are vaguely "trusted". We need to know what they do in more precise details.

The End of Trust in the Web. The World Wide Web evolved rapidly in 90's and the notion has changed from a network to a platform where people migrate desktop applications. The security model of the current version of the web is based on an assumption that the good guys develop their application, expose it on the web, and then let other good guys using it while stopping bad guys from misusing it.

The business trend of outsourcing processes or the construction of virtual organizations have slightly complicated this initially simple picture. Now running a "service" means that different service (sub)components can be dynamically chosen and different partners are chosen to offer those (sub)services. Hence we need different trust establishment mechanisms (see e.g. [10]).

This assumption is no longer true for the new world of Web 2.0 and the Future Internet. Even now a user downloads a multitude of communicating applications ranging from P2P clients to desktop search engines, each of them ploughing through the user's platform, and springing back with services from and to the rest of the world. To deal with the untrusted code either .NET or Java can exploit the mechanism of permissions. Permissions are assigned to enable execution of potentially dangerous or costly functionality, such as starting various types of connections. The drawback of permissions is that after assigning a permission the user has very limited control over how the permission is used. Conditional permissions that allow and forbid use of the functionality depending on such factors as bandwidth or the previous actions of the application itself (e.g. access to sensitive files) are also out of reach. Once again the consequence is that either applications are sandboxed (and thus can do almost nothing), or the user decided that they are trusted and then they can do almost everything.

The mechanism of signed assemblies from trusted third parties does not solve the problem either. Currently a signature on a piece of code only means that the application

* Research partly supported by the Projects EU-FP6-IST-STREP-S3MS, EU-FP6-IP-SENSORIA, and EU-FP7-IP-MASTER. We would like to thank Eric Vetillard for pointing to us the domain of Next Generation Java Card as the Challenge for the Future Internet.

comes from the software factory of the signatory, but there is no clear definition of what guarantees it offers. It essentially binds the software with nothing. We built our security models on the assumption that we could trust the vendors (or at least some of them). The examples from reputable companies such as Channel 4 (or BBC, Sky TV etc.) show that this is no longer possible. Still we really want to download a lot of software.

The Smart(Card) Future of the Web. The model that we have described above is essentially the web of the personal computers. None of the users complaining about 4oD [14] have considered their PC or their Web platform "broken" because it allowed other people to make use of it. They did not consider returning their PC for repair. They considered themselves being gullible users ripped off by an untrusted vendor.

Another domain at the opposite side of the psychological spectrum is smartcard technology. The technology enjoyed worldwide deployment in 90's with Java Card Applets and their strict security confinement. At the beginning of the millennium, many applications such as large SIM cards and identity management businesses are implemented on smart-cards to address mobile devices security challenges.

The smartcard technology evolved with larger memories, USB and TCP/IP support and the development of the Next-Generation Java Card platform with Servlet engine. The Future Internet will be composed by those embedded Java Card Platforms running on high end smart cards in our phones and our wallets, each of them connecting to the internet and performing secure transactions with distributed servers and desktop browsers without complicated middleware or special purpose readers.

We still want to download a huge amount of software on our phones but there is a huge psychological difference from a consumer perspective. If our PC is sluggish in responding, *we* did something wrong or downloaded the wrong software, if our phone is sluggish, *it* is broken. Moreover, in the realm of next generation Java card platforms we cannot just download a software without knowing what it does. The smart card web platform must have a way to check what is downloading.

2 Security by Contract for the Smart Future Internet

In the previous FLACOS workshop we [11] have proposed the notion of Security-by-Contract (S×C)[5, 4]. In S×C we augment mobile code with a claim on its security behavior (an *application's contract*) that could be matched against a mobile *platform's policy* before downloading the code. A digital signature does not just certify the origin of the code but also bind together the code with a contract with the main goal to provide a semantics for digital signatures on mobile code. This framework is a step in the transition from trusted code to trustworthy code.

S×C *Workflow.* At *development time* the mobile code developers are responsible for providing a description of the security behavior that their code finally provides. Such a code might also undergo a formal certification process by the developer's own company, the smart card provider, a mobile phone operator, or any other third party for which the application has been developed. By using suitable techniques such as static analysis, monitor in-lining, or general theorem proving, the code is certified to comply with the

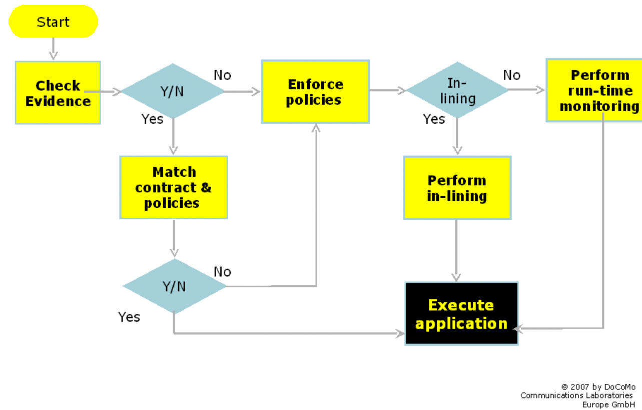

Fig. 1. SxC Workflow

developer's contract. Subsequently, the code and the security claims are sealed together with the evidence for compliance (either a digital signature or a proof) and shipped for deployment. At *deployment time*, the target platform follows a workflow similar to the one depicted in Fig.1 (see [19]). First, it checks that the evidence is correct. Such evidence can be a trusted signature or a proof that the code satisfies the contract (one can use Proof-Carrying-Code (PCC) techniques to check it.

As we have evidence that the contract is trustworthy, the platform checks, that the claimed policy is compliant with the policy that our platform wants to enforce. If it is, then the application can be run without further ado. It is a significant saving from in-lining a security monitor. In case that at *run-time* we decide to still monitor the application then, as with vaccination, we inline a number of checks into the application so that any undesired behavior can be immediately stopped or corrected.

Contract for the Smart Future Internet. A *contract* is a formal complete and correct specification of the behavior of an application for what concerns relevant security actions (Virtual Machine API Calls,Web Messages etc). By signing the code the developer certifies that the code complies with the stated claims on its security-relevant behavior. A *policy* is a formal complete specification of the acceptable behavior of applications to be executed on the platform for what concerns relevant security actions.

Technically, a contract can be a security automaton in the sense of Schneider [8], and it specifies an upper bound on the security-relevant behavior of the application: the sequences of security-relevant events that an application can generate are all in the language accepted by the security automaton. We can have a slightly more sophisticated approach using Büchi automata [18] if we also want to cover liveness properties that can be enforced by Edit automata. This definition can be sufficient for theoretical purposes but it is hardly acceptable for any practical use.

A variant of the PSLANG language [1] has been proposed for S×C for mobile code (.NET and Java). The formal counterpart of the language is the notion of *automata modulo theory* [12] where atomic actions are replaced by expressions that can finitely

capture infinite values of API parameters. For the smart future internet, we need to identify a suitable language for the specification of contracts and policies at a level of abstraction that is suitable and can be used for *all* S×C phases (Fig.1)

Application-contract compliance. Static analysis can be used at development time to increase confidence in the contract. With static analysis, program analysis and verification algorithms are used in an attempt to *prove* that the application satisfies its contract.

The major advantage of static analysis is that it does not impose any runtime overhead, and that it shows that all possible executions of a program comply with the contract. The major disadvantage is that the problem of checking application-contract compliance is in general undecidable, and so automatic static analysis tools will typically only support restricted forms of contracts, or restricted forms of applications, or the tool will be *conservative* in the sense that it will reject applications that are actually compliant, but the tool fails to find a proof for this.

The programs and services running on the embedded servlet will be significantly more complex and have actions at different level of abstractions whose full security implications can be understood by considering all abstraction levels at once. The challenges for static analysis is that with expressive notions of security contracts, verifying application-contract compliance is actually as hard as verifying compliance with an arbitrary specification [16]. Moreover, contracts for applications in the Smart Future Internet will have a complexity that is comparable to the level of abstractions of current concurrent models that are used for model checking hardware and software systems (in 10^{10} states or transitions and beyond).

A standard approach to make program verification and analysis algorithms scale to large programs is to make them *modular* of the program independently. This is particularly hard for application-contract compliance checking, because the security state of the contract is typically a global state, and the structure of the contract and its security state might not align with the structure of the application. Annotations are required on *all* methods to specify how they interact with the security state, and not only on methods that are relevant for the contract at hand. This annotation overhead is prohibitive, so a key challenge is to look for ways to reduce the annotation burden. An interesting research question is whether a program transformation (similar to the security-passing style transformation used for reasoning about programs sandboxed by stack inspection [17]) can improve this situation.

A second approach to address scalability is to give up soundness of the analysis, and to use the contract as a model of the application in order to generate security tests by applying techniques from Model Based Testing [20]. Losing soundness is a major disadvantage: an application may pass all the generated tests and still turn out to violate the contract once fielded. However, the advantages are also important: no annotations on the application source code are needed, and the tests generated from the contract can be easily injected in the standard platform testing phase, thus making this approach very practical. A challenge to be addressed here is how to measure the coverage of such security tests. When are there enough tests to give a reasonable assurance about security? It is easy to automatically generate a huge amount of tests from the contract. Hence it is important to know how many tests are sufficient, and whether a newly generated test increases the coverage of the testing suite.

Matching Contract and Policy on the Smart Future Internet. We must show that the behavior described by the contract is acceptable according to our platform policy. The operation of matching the application's claim with the platform policy requires that the contract is trustworthy, i.e. the application and the contract are sealed together with a digital signature when shipped for deployment or by shipping a proof that can checked automatically. A simple solution is to build upon automata theory, interpret contract and policy as automata and use language inclusion . Given two such automata Aut^C (representing the contract) and Aut^P (representing the policy), we have a match when the language accepted by Aut^C is a subset of the language accepted by Aut^P.

Once the policy and the contract are represented as automata then one can either use language inclusions [12] or simulation [13] to check whether the contract is acceptable according our platform policy. This solution is only partial because the automata that we have envisaged do not store the values of the arguments of allowed/disallowed APIs. In order to do this Contracts and policies for the future internet must be history-dependent: the arguments of past allowed actions (API calls, WS invocations, SOAP messages) may influence the evolution of future access control decision in a policy.

Further, in our current implementation of the matcher that runs on a mobile phone, security states of the automata are represented by variables over finite domains e.g. smsMessagesSent ranges between 0 to 5. [1, 2]. A possible solution could be to extend the work on finite-memory automata [9] by Kaminski and Francez or other works [15] that studied automata and logics on strings over infinite alphabets.

An approach to address scalability is to give up soundness of the matching and use algorithms for simulation and testing. A challenge to be addressed is how to measure the coverage of approximate matching. Which value should give a reasonable assurance about security? Should it be an absolute value? Should it be in proportion of the number of possible executions? In proportion to the likely executions? An interesting approach could be to recall to life a neglected section on model checking by Courcoubetis et al [3] in which they traded off a better performance of the algorithm in change for the possibility of erring with a small probability.

Inlining a monitor on Future Internet Applications. What happens if matching fails? or what happens if we do not trust the evidence that the code satisfies the contract? If we look back at Fig.1 monitor inlining of the *contract* can provide strong assurance of compliance. With *monitor inlining* [7], code rewriting is used to push contract checking functionality into the program itself. The intention is that the inserted code enforces compliance with the contract, and otherwise interferes with the execution of the target program as little as possible. Monitor inlining is a well-established and efficient approach [6] however a major open question is how to deal with concurrency efficiently.

Servents in the Smart Future Internet will need to monitor the concurrent interactions of tens of untrusted multithreaded programs. An inliner needs to protect the inlined security state against race conditions. So all accesses to the security state will happen under a lock. A key design choice for an inlining algorithm is whether to lock across security relevant API calls, or to release the lock before doing the API call, and reacquiring it when the API call returns.

The first choice (locking across calls) is easier to get secure, as there is a strong guarantee that the updates to the security state happen in the correct order. This is much

trickier for an inliner that releases the lock during API calls. However, an inliner that locks across calls can introduce deadlocks in the inlined program, because some of the security relevant API calls will themselves block. And even if it does not lead to deadlock, acquiring a lock across a potentially blocking method call can cause serious performance penalties. A partial solution is by partitioning the security state into disjoint parts, and replacing the global lock, by per-part locks. This improves efficiency, but depending on application and policy, it can still introduce deadlocks. The challenge is how to inline a monitor into a concurrent program so that it cannot create a deadlock in future interactions with other unknown programs yet to be downloaded.

The ability to resist to changes in context (i.e. new concurrent programs downloaded after the inlined program) is essential for usability. The inlined version of 4oD should not get in the way if later on I want to download a (inlined) role-playing game. It is possible that two malicious software downloaded at different instants try to cooperate in order to steal some data. The security monitor should be able to spot them but not be deadlocked by them. If inlining is performed by the code producer, or by a third party, the code consumer (client that runs the application) needs to be convinced that inlining has been performed correctly. Without a secure transfer of the guarantees of application-contract compliance to the client, it is easy for an attacker to modify either the application or the contract, or for an application developer to lie about the contract.

Cryptographic signatures by a trusted (third) party is a first solution even if it transfer the risk from the technical to the legal domain. The trusted party vouches for application-contract compliance. Note the difference with the use of signatures in the traditional mobile device security model. In the security-by-contract approach, a signature has a clear semantics [5]: the third party claims that the application respects the supplied contract. Moreover, what is important is the fact that the decision whether the contract is acceptable or not remains with the end user. If an application claims that it will not connect to the internet and instead it does, at least you can bring the signatory to the court for fraudulent commercial claims.

Another solution is whether we can use the techniques PCC for this. In PCC, the code producer produces a proof that the code has certain properties, and ships this proof together with the code to the client. By verifying the proof, the client can be sure that the code indeed has the properties that it claims to have.

The difficulty of the endeavour is that the code has not been produced to be verified compliant against a security property but usually to actually do some business. In other words, the code producer is not aware of the property and the property producer is not aware of the code. In this scenario verification is clearly an uphill path.

When we inline a contract we know precisely what code we are inlining and also what property the inlined code should satisfy. So, we can ask the inliner to do this automatically for us and ask them to generate the proof directly. This should make it relatively easy to check that code complies with the contract: the generation of a proof should be easier, and the size of the proof would also be acceptable for inlined programs. The challenge is to identify automatic inlining mechanisms that inline a monitor for a security contract and generate an easily checkable proof for industrial applications in the Smart Future Internet.

References

1. I. Aktug and K. Naliuka. Conspec - a formal language for policy specification. Proc. of the 1st Int. Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM2007), 2007.
2. N. Bielova, M. Dalla Torre, N. Dragoni, and I. Siahaan. Matching policies with security claims of mobile applications. In *Proc. of the 3rd Int. Conf. on Availability, Reliability and Security (ARES'08)*. IEEE Press, 2008.
3. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in Sys. Design*, 1(2-3):275–288, 1992.
4. L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .net platform. *Elsevier Inform. Sec. Technical Report*, 13(1):25–32, 2008.
5. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *Proc. of the 4th European PKI Workshop Theory and Practice (EUROPKI'07)*. Springer-Verlag, 2007.
6. U. Erlingsson and F.B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. of the 1999 New Security Paradigms Workshop (NSPW'99)*.
7. U. Erlingsson and F.B. Schneider. IRM enforcement of Java stack inspection. In *Proc. of the 2000 IEEE Symp. on Security and Privacy*, pages 246–255. IEEE Computer Society, 2000.
8. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *TOPLAS*, 28(1):175–205, 2006.
9. M. Kaminski and N. Francez. Finite-memory automata. *Theor. al Comp. Sci.*, 134(2):329–363, 1994.
10. Y. Karabulut, F. Kerschbaum, F. Massacci, P. Robinson, and A. Yautsiukhin. Security and trust in it business outsourcing: a manifesto. In S. Etalle and P Samarati, editors, *Proc. of the 2nd Int. Workshop on Security and Trust Management (STM'06)*, ENTCS. Elsevier, 2006.
11. F. Massacci, N. Dragoni, and I. Siahaan. A Security-by-Contracts Architecture for Pervasive Services. 2007.
12. F. Massacci and I. Siahaan. Matching midlet's security claims with a platform security policy using automata modulo theory. In *Proc. of The 12th Nordic Workshop on Secure IT Systems (NordSec'07)*, 2007.
13. F. Massacci and I. Siahaan. Simulating midlet's security claims with automata modulo theory. In *Proc. of the 2008 workshop on Prog. Lang. and analysis for security*, 2008. submitted.
14. CNET Networks. Channel 4's 4od: Tv on demand, at a price. *Crave Webzine*, January 2007.
15. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *TOCL*, 5(3):403–435, 2004.
16. F.B. Schneider. Enforceable security policies. *ACM Trans. on Inf. and Sys. Security*, 3(1):30–50, 2000.
17. J. Smans, B. Jacobs, and F. Piessens. Static verification of code access security policy compliance of .net applications. *J. of Object Technology*, 5(3):35–58, 2006.
18. C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Inform. and Comp.*, 206(2-4):158–184, 2007.
19. D. Vanoverberghe, P. Philippaerts, L. Desmet, W. Joosen, F. Piessens, K. Naliuka, and F. Massacci. A flexible security architecture to support third-party applications on mobile devices. In *Proc. of the 1st ACM Comp. Sec. Arch. Workshop*, 2007.
20. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *Proc. of the 10th Eur. Software Eng. Conf. held jointly with 13th ACM SIGSOFT Int. Symp. on Found. of Software Eng.*, pages 273–282. ACM Press, 2005.