A Reference Architecture for Software Protection

Bjorn De Sutter
Ghent University
Belgium
bjorn.desutter@ugent.be

Paolo Falcarin
University of East London
UK
falcarin@uel.ac.uk

Brecht Wyseur
NAGRA
Switzerland
brecht.wyseur@nagra.com

Cataldo Basile
Politecnico di Torino, Italy
cataldo.basile@polito.it

Mariano Ceccato
Fondaz. Bruno Kessler, Italy
ceccato@fbk.eu

Jerome d'Annoville
Gemalto, France
jerome.d-annoville@gemalto.com

Michael Zunke
SafeNet
Germany
michael.zunke@safenet-inc.com

***Abstract*—This paper describes the ASPIRE reference architecture designed to tackle one major problem in this domain: the lack of a clear process and an open software architecture for the composition and deployment of multiple software protections on software applications.**

Keywords— Security services, client-server architecture, software protection tool chain, security requirements, composability

I. Introduction and Motivation

Software protection has been an intrinsic problem of software engineering since software became a commercial product. It is crucial to mitigate attacks such as reverse engineering, piracy, and tampering [1]. In general, software protection research aims at developing algorithms to protect the integrity of data and applications deployed on untrusted devices. Software protection's scope spans a range of different heterogeneous research topics: obfuscation and cryptography [2], digital rights management [3], information hiding [4], reverse engineering [5], compilers and code transformations [6], and distributed systems [7].

Recent trends increase end-user demand to use mobile devices for a variety of applications that were until now limited to secured devices such as set-top boxes, online license servers, and desktops apps with USB dongles. The growing zoo of mobile devices makes it inconvenient to require application-specific security hardware, such as smart-cards or USB dongles: all offerings need to work on top of any (open) platform the user wants to use. Software-based software protection that can guarantee secure application execution has therefore become utterly important. It can be a maker and a breaker in domains like multi-screen mobile TV, software licensing, and credentials and sensitive data stored on mobile devices. To protect their assets, stakeholders in mobile devices, mobile services and mobile software need trustworthy and affordable software-based security solutions.

In the European FP7 project ASPIRE [8], three leaders in security solutions team up with four academic partners. Gemalto is the world leader in the smart card business. SafeNet is the world leader in token-based software licensing. Nagra is the world's leading supplier of end-to-end security solutions for set-top box TV operators. All three of them are looking for ways to extend their product portfolios with more software-based protections. Such software protections typically consist of components injected into software and of transformations applied to the original, non-protected software to invoke the protections. A non-trivial design problem is how to integrate multiple protections. With current tools, deploying software protections is either a cumbersome manual process or an "all or nothing" approach, where the use of one protection tool prevents the use of complementary tools. For example, Metaforic is applied on the source code [9], with a mandatory finalization step on the binary that prevents any further protection tool from working. Cloakware requires extensive manual integration [10], making the implementation costs a substantial investment in time and effort, and often even requiring consultancy. The real world proves, however, that the required investment in security is typically underestimated, resulting in refusal to deal with required changes in the build process or not investing the time for understanding the available tools.

This paper presents results of ASPIRE's R&D into a plugin-based software protection tool chain and a corresponding software architecture that support the combined deployment of a wide range of software protections developed largely independently from each other. The protections can be deployed on software assets by means of source-code annotations, which greatly helps to increase the productivity of the software developer. We describe the considered security requirements, the ASPIRE software architecture designed to ease the integration and deployment of protections, and the ASPIRE Compiler Tool Chain (ACTC) designed to compose the protections.

II. Security Assets and Requirements

The scope of the ASPIRE project entails protections against Man-At-The-End (MATE) attacks [11]. Their targets are assets embedded in natively compiled software stored and running on completely untrusted devices under full control of the attackers. Unless the vendors adequately protect their software, attackers (thank to tools such as disassemblers, de-compilers, and debuggers) have mostly complete access to the embedded assets. The considered assets are private or public and unique or global data, such as all kinds of cryptographic keys, credentials, and privacy-sensitive data; traceable code/data, such as watermarks; as well as application code including software IP and security libraries and application execution, i.e., the functional behavior of an application, rather than its representation.

Attackers can target the assets through static attacks, in which they study and manipulate code and data before the software is executed, or through dynamic attacks, in which they study and manipulate code and data during the software's execution. Hybrid attacks, that combine static and dynamic aspects, are possible as well. With respect to static techniques, ASPIRE considers attacks involving static structural code and data recovery methods, structural matching methods of multiple binaries, and static tampering attacks. With respect to dynamic techniques, ASPIRE considers attacks involving com-

munication channels (spoofing, sniffing, replay), fuzzing, debugging, dynamic structure and data analysis, and dynamic tampering attacks such as code injection.

Providing full protection against such attacks is impossible, as the attacker is assumed to have full access and control. Our goal is hence to delay an attacker, to increase his required investment for identifying an attack path, and to make it harder to exploit an identified path in a profitable manner. To that extent, it is necessary to combine multiple lines of defense: combinations of many forms of protections that not only protect the original assets in the application, but also each other. In a sense, the correct deployment of a protection becomes an asset itself that needs just as much protection in order to resist determined attackers. ASPIRE proposes five lines of defense: data hiding (data obfuscation and white-box crypto), code hiding (code obfuscations, instruction set customization, server-side execution, and on-the-fly code downloading), anti-tampering (code guards and anti-debugging), remote attestation, and renewability. The latter denotes technique to generate not one, but multiple, renewed software instances. It includes spatial diversification to limit the scale at which an attack path can be exploited, and temporal diversification to limit the attacker's window of opportunity to exploit an attack path. In an application, all the protections combined together should ideally protect all potential attack paths on all the embedded assets. As most forms of protection come with a significant run-time overhead and possibly with significant impact on the software development life cycle (SDLC), developers also have to trade off the provided protection for other criteria.

In this scope, an initial set of requirements was elicited from three industrial use cases: a DRM library for Android's DRM framework, a software license manager for Android Dalvik apps, and a one-time password generator. The resulting requirements were then generalized to serve a wider list of applications. We specified three types of security requirements.

Non-functional security requirements are properties the protected application must possess. They specify how the application needs to behave and impose constraints on how the application needs to be protected. If the unprotected application does not meet them with respect to the embedded assets, the ASPIRE compilation infrastructure will need to ensure that the implementation is modified or compiled such that the protected application comprises these properties. The non-functional security requirements include confidentiality, integrity, privacy, non-repudiation, and execution correctness.

The validation of these requirements is challenging, as many requirements lack a commonly agreed validation methodology. In the ASPIRE project, we try to combine attack modelling, software complexity metrics (incl. metrics to measure resilience against tools in the attacker's tool box), and security audits by industry security experts to address this issue.

Functional security requirements are security services that need to be provided by the application and its supporting ecosystem. With the latter, we define the complete system setup that includes a Trusted Entity. With this Trusted Entity, we capture the entity that hosts security services such as the remote attestation verification, the distribution of updates, etc. In most cases, the Trusted Entity is controlled by the Application Vendor and its role is to ensure trustworthy execution of the ASPIRE-protected (client-side) application. Functional security requirements are system requirements that emerge from some security goals, and express which services need to be implemented. If these services are not present in the original (unprotected) application, the ASPIRE compilation infrastructure needs to inject them and enforce their deployment. These security requirements are easy to validate, because the presence of the services in the application can easily be verified. Whereas non-functional security requirements can differ for multiple assets in the same application, the functional requirements are typically whole-application requirements.

Software security assurance is the process of ensuring that software is developed and maintained to operate at a level of security that is consistent with the security goals and other requirements of the SDLC. Hence, **assurance security requirements** describe activities during the entire SDLC to assure that the application is not subject to vulnerabilities. These requirements relate to the build process and the compilation infrastructure itself, in contrast to the previous two classes of requirements, which relate to the target application.

One of the critical assurance security requirements in ASPIRE is the plug-in based nature of its ACTC. This stems on the one hand from the need to reuse existing public or proprietary infrastructure where possible to lower the entry ticket price of software protection, but to support combining that infrastructure with in-house developments on the other hand, to allow its users to differentiate themselves and to support security through obscurity, which today remains popular in practice.

Given the need for multiple lines of defense to protect multiple assets with different requirements, a protection tool chain clearly needs to support the composability of many protections, both at a fine-grained granularity for deploying multiple protections on the same code fragment or data structure, as on a coarse-grained granularity to apply multiple protections on the same application), and in functional compositions where one protection is applied to the code implementing another one.

Our goal is hence to develop a protection tool chain that supports the integration of different protections developed by different partners in a plug-in fashion, and that facilitates their composed deployment through a unique interface consisting of source code annotations that mark the assets in the code, the threats on those assets, and the protections to be deployed. To achieve this goal, there is a clear need to define a reference protection architecture that documents how techniques operate and compose, and the common infrastructure they can build on. For that reason, the ASPIRE consortium investigated considerable effort in the ASPIRE Reference Architecture.

III. The ASPIRE Reference Architecture

As a basis, we selected the multi-tier architecture structure depicted in Fig. 1. This captures the case where a multitude of client applications connect to a portal infrastructure, which manages the connection to a set of backend servers. The ensemble of the portal infrastructure and backend servers is denoted as the *ASPIRE security server*. We adopt such infrastructure for the client-server communication of our network-based

protections, and deploy this in parallel to the client-server communication that the original application might already use.

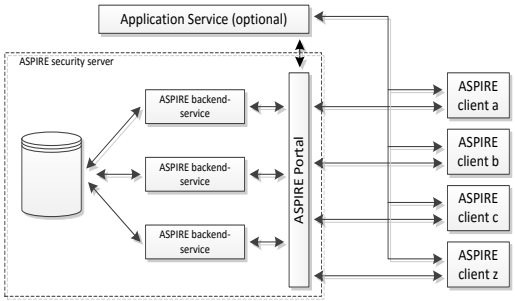

Fig. 1. ASPIRE multi-tier architecture: high-level view.

This approach was selected for many reasons. Firstly, the co-existence of the ASPIRE client-server communication and the original client-server communication minimizes the impact on existing application services. This eases the real-world deployment of protections in many scenarios, for example when time-to-market constraints are met by gradually deploying more protections over time. Additionally, this co-existence minimizes the dependencies between original client-server communications and ASPIRE communications. This is necessary, because of the wide range of applications that ASPIRE targets, of which no a priori assumptions regarding their communications can be exploited. The complete co-existence allows the protections and the online security services to operate on applications that originally were offline applications, or in scenarios where the existing communication cannot be leveraged, e.g., when one-directional satellite communication serving live video cannot be exploited for remote attestation. Last but not least, the application service and the ASPIRE protection service may be the responsibility of different entities and may be running in different server infrastructure facilities.

While we strive for minimal impact on the original client-server communication, we also support interaction between the original server (if any) and the security services, through the ASPIRE portal. For example, the application server can ask an ASPIRE backend service to obfuscate a key for a particular protected client instance and to send that key. Another interesting use of this interface is an application server requesting a trustworthiness status on particular clients, upon to let the server decide if and how to proceed with the original service.

Secondly, a multi-tier architecture to support the ASPIRE protections was selected because its flexibility, scalability, and reusability. A portal service serves as a terminator for the secure link between protected applications and the server-side; the individual protection services do not need to take the communication protocol details into account. This portal would be a lightweight service that re-directs messages between protected applications and the relevant protection services. As such, protection services can be scaled onto different devices. This supports the adoption of the ASPIRE results in an industrial context. Additionally, this also facilitates concurrent development of protection techniques within the ASPIRE consortium, as protection services can run completely independently and even be embodied in any given form, such as a script, a local process, or a service on a different physical machine – as long as the ASPIRE portal knows how to communicate with them. This logic abstracts the communication for the protection techniques, as shown in Fig.2, to ease their individual development.

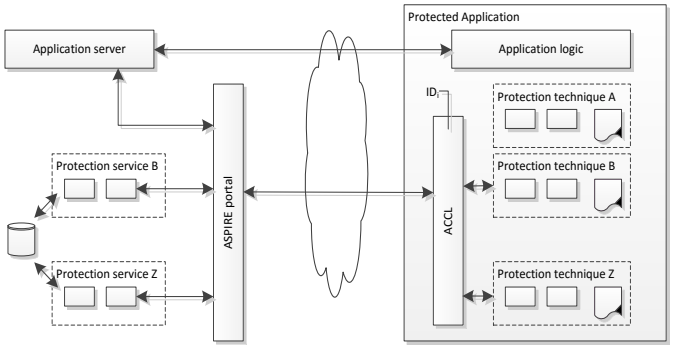

Fig. 2. ASPIRE architecture view on single client-server.

To facilitate the communication between the client-side protection technique components and the corresponding server-side support components via the ASPIRE portal (implemented on Nginx [12]), we introduce a special-purpose communication logic: the ASPIRE Client-side Communication Logic (ACCL), relying on curl native HTTP library [13] and on OpenSSL [14] to protect the communication from Man-in-the-Middle attacks.

As for the protocol, the most important aspect is its high-level behaviour, and in particular how different communications and services are initiated and invoked. From a practical point of view, the best approach would be one in which the individual protected applications take the initiative to query the portal, and where the server-side response then depends on a stateless computation. Indeed, this approach can support many business models as it easily scales due to its statelessness, and as it is independent of the client-side network infrastructure. For example, clients can easily communicate with an HTTP portal while being behind a firewall or while hopping between different networks such as 3G and different Wi-Fi networks.

Along with the simple client-server interaction pattern, we also allow active bi-directional communication between the ASPIRE portal and the communication logic in protected client applications, by using WebSocket [15] . Using this technology, clients can initiate a channel with the ASPIRE portal, which the portal can use at any time during the lifetime of that channel to invoke client-side operations when it wants.

Finally, the ACTC injects a unique ID into each generated instance of the software. The ASPIRE database keeps track of the valid IDs in use, such that the server can deliver the correct security services for each running instance.

IV. The ASPIRE Tool Chain

As shown in Fig. 3, the ACTC comprises (i) plug-ins that rewrite **source code** to deploy protections and that generate additional code to implement certain protections, (ii) a **standard compiler and linker** that can link in additional protection libraries such as the communication libraries discussed in the previous section, and (iii) plug-ins that rewrite all of the **binary code**. The ACTC produces a set of protected binaries and relevant data for deploying the security services: one application

for the client mobile device, and one (or more) for the server if applicable. Source-level plug-ins are implemented as rewrite rules for TXL [16]. They are invoked in a well-chosen order, with each of them taking as input and producing as output the (partially protected) software in the form of pre-processed C or C++ code. This facilitates the integration of new plug-ins. Binary-level plug-ins are developed as components of the open source link-time binary code rewriting framework Diablo [17]. In this framework, an internal representation of the application (a binary or shared library) and of the linked-in protection libraries is built, on which the different protection plug-ins are then applied in sequence, each of them again producing a complete program representation. By letting each plug-in produce a full program representation available for manipulation by the next plug-in, we ensure that later protections can be deployed on protection components generated for earlier protections. Depending on their nature and synergies with other protections, some protections are better deployed during the source-level rewriting, during the binary-level rewriting, or in a combination of both. The ability to use any standard-compliant compiler, like GCC or LLVM, and the ability to integrate their own protection plug-ins are important assurance requirements of the project's industrial partners.

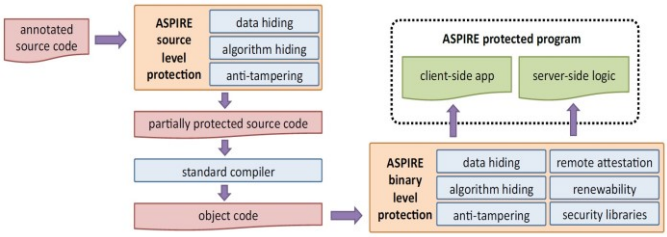

Fig. 3. Abstract view of the ASPIRE Tool-Chain process.

By means of well-known, standard compiler features such as pragmas and attributes, the ACTC offers a convenient method for developers to annotate and the assets in their software, to which the software protections will then be applied automatically. This allows for a clean separation of software protections from the logic of the software application. The specification of the annotations will be published in 2016. The annotations are extracted from the source code before compilation, and passed to the link-time rewriter, which identifies the annotated code regions in the binary code by means of standard DWARF [18] debug information in the compiled object files. Again, this favours deployment and integration in the real world.

The annotations and the ACTC JSON-based configuration features allow the ACTC's users to select which protections to apply where. Furthermore, the necessary configuration options are available to describe composability constraints. An ACTC super-user or developer can set the latter configuration once during the integration of his plug-ins, after which the ACTC will automatically inform ordinary tool chain users (i.e., application developers) about impossible compositions of protections. This helps them to balance conflicting requirements and to make a trade-off taking into account composability and phase-ordering issues, as well as protection requirements.

V. Conclusions and Future Work

We described the ASPIRE reference architecture, the underlying requirements, and the ACTC to ease the composition and the deployment of software protections. Within the ASPIRE project, the application of the tool chain and the reference architecture has been validated successfully on three industrial user cases. Like the iterative design of the ASPIRE tool chain, the full ASPIRE architecture, which details how all ASPIRE protections operate and fit in the discussed reference architecture, is publicly available online [8].

In the rest of the project, we will also validate the provided protection strength by means of security audits, and publicly demonstrate our contributions. We are also developing automated decision support to help users of the ACTC with the selection and deployment of the best combinations of protections, given their assets and SDLC constraints.

Acknowledgment and Note

The ASPIRE project has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 609734. During the course of the project, Gemalto acquired SafeNet; the problem to solve and the selected use cases remained unchanged.

References

[1] C. Collberg, J. Nagra. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison Wesley, 2009.

[2] S. Garg, S., C. Gentry, et al. Candidate indistinguishability obfuscation and functional encryption for all circuits. In proc. IEEE Annual Symposium on Foundations of Computer Science, 2013, 40–49.

[3] W. Zeng, H. Yu, and C.-Y. Lin, eds. Multimedia security technologies for digital rights management. Vol. 18. Academic Press, 2011.

[4] Brecht Wyseur, "White-box cryptography: hiding keys in software." NAGRA Kudelski Group (2012).

[5] T.E. Dube, B.D. Birrer, et al. Hindering Reverse Engineering: Thinking Outside the Box, IEEE Security & Privacy, 2008, Vol 6(2), 58–65

[6] B. Coppens, B. De Sutter, J. Maebe, Feedback-Driven Binary Code Diversification, ACM Transactions on Architecture and Code Optimization, Vol. 9 Nr. 4, Art. 24, January 2013.

[7] R. Scandariato, Y. Ofek, P. Falcarin, and M. Baldi, “Application-Oriented Trust in Distributed Computing,” in Proc. 2008 Third Int'l Conf. on Availability, Reliability and Security. IEEE, 2008, 434–439.

[8] EU FP7 ASPIRE project (Advanced Software Protection: Integration, Research and Exploitation). Online at http://www.aspire-fp7.eu/

[9] Metaforic product information.Online at http://metaforic.com/products .

[10] Irdeto information about ActiveCloak. Online at http://irdeto.com/products/products.html

[11] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski. "Guest Editors' Introduction: Software Protection." Software, IEEE 28(2), 2011: 24–27.

[12] Nginx web server, On-line at http://nginx.org

[13] Curl open source library, On-line at http://curl.haxx.se

[14] OpenSSL project. On-line at https://www.openssl.org/

[15] Internet Engineering Task Force (IETF), “The WebSocket Protocol”, RFC 6455, December 2011, http://tools.ietf.org/html/rfc6455

[16] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow, "Txl: A rapid prototyping system for programming language dialects", Computer Languages, vol. 16, no. 1, pp. 97–107, 1991.

[17] Diablo project. On-line at http://diablo.elis.ugent.be/

[18] The DWARF Debugging Standard. On-line at http://dwarfstd.org/