Remote Trust with Aspect-Oriented Programming

Paolo Falcarin, Riccardo Scandariato, Mario Baldi
Politecnico di Torino, Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, Torino, Italy
{paolo.falcarin, riccardo.scandariato, mario.baldi}@polito.it

Abstract

Given a client/server application, how can the server entrust the integrity of the remote client, albeit the latter is running on an un-trusted machine? To address this research problem, we propose a novel approach based on the client-side generation of an execution signature, which is remotely checked by the server, wherein signature generation is locked to the entrusted software by means of code integrity checking. Our approach exploits the features of dynamic aspect-oriented programming (AOP) to extend the power of code integrity checkers in several ways. This paper both presents our approach and describes a prototype implementation for a messaging application.

1. Introduction

Security has always been a primary concern for industry, and recently the interest for client-side software protection has grown. In general, network-enabled software suffers from some inherent security problems, like unauthorized modification by either malicious individuals (host machine cannot be trusted because of the user threat) or digital entities, like viruses and Trojan horses (host machine cannot be trusted because of the network threat). Many circumstances do exist in which it is necessary to protect software from malicious modifications once it is delivered to the public, e.g., e-voting and e-commerce systems.

In particular, this work has been carried out in the context of a research project aiming at providing an answer to the following research question: How can be a client application entrusted albeit running on an un-trusted machine? With the term "un-trusted machine" we mean a networked computing base (e.g., networked computers and mobile devices) in which a possibly malicious user has complete (conceivably physical) access to system resources (e.g., memory and disks) and tools (e.g., debuggers) in order to reverse-engineer the application code. In the scope of this paper, an application is deemed trusted whenever its executed code has not been altered prior to and *during* execution. Satisfying such integrity requirement in a hostile environment is a challenging issue and TrustedFlow is the all-in-software solution we present to tackle the problem [1].

In our approach, clients emanate a continuous flow of idiosyncratic tags towards to server. The tags flow is generated by a software module that is securely combined with the original application. As far as the application code remains genuine, the module will produce valid tags, which are used as continuous evidence to the remote server that the client code is authentic. Note that disabling the module is not an option for the attacker. In such case, tags would no longer be produced and the server would notice the attack attempt immediately. This work presents prototypal implementation of TrustedFlow based on the deployment and nonstop replacement of integrity-checking modules implemented as dynamic aspects. As further discussed in the related-work section, traditional integrity self-checking techniques offer no guaranty that self-check has been actually performed. As a major contribution, our approach provides remote verification of tags in order to verifying that the check has been duly preformed.

Our solution can be deployed in several usage scenarios. In particular, it can be used to restrict access to a public server, to let in genuine clients only, possibly distributed by the server itself. Examples of existing applications are Yahoo services (Yahoo advanced services are available only to users deploying Yahoo's client), gambling servers, and on-line submission of final exams. Along these lines, a messaging system, composed of a server acting as entrusting entity and a client acting as entrusted entity, has been developed as a proof of concept.

2. Related Work

In general, tamper resistance is pursued by means of a set of methodologies aimed at protecting software from unauthorized modification, distribution, and use. Among the several possible attacks, we focus on integrity attacks, i.e., those aiming at tampering with

application code for malicious purposes, like bypassing security, infringing licenses, or forcing different execution (e.g., to manipulate ballots in e-voting applications).

Different solutions have been proposed in literature to protect software from above-mentioned rogue behaviors, being *integrity self-checking* the main representative.

Most of commercial applications just rely on static self-checking, in which the program checks its integrity only once, during start-up (e.g. to check license code), while current research is focusing on dynamic self-checking, in which the program verifies regularly its integrity during run-time. The key issue is to avoid that the self-checking function itself is removed or disabled, without being detected. To this aim, networks of integrity-checkers (called guards) were proposed to detect changes to binary code [2]. Protection against code modification is enforced by including a large number of guards, each protecting a fraction of the application. Clearly, the task of finding and disabling all guards is significantly more difficult. A similar approach provides a mechanism to redundantly test for changes [3].

Chen et al. [4] compute a fingerprint of application code on the basis of its actual execution. This method makes it possible to thwart attacks using automatic program analysis tools and other static methods.

Obfuscation aims at increasing the attack complexity by making it hard for the attacker to comprehend the behavior of a decompiled program [5]. Obfuscation techniques are based on the addition of complexity to source code structure (without changing its behavior) through different kinds of code transformations. Code obfuscation transformations are also employed to hide a tamper resistance code embedded in the software so that it cannot be easily detected and removed. However, in most cases, breaking obfuscation it is just a matter of time and attacker's skills [6], and the overhead of obfuscation techniques can be significant both in terms of code size increase and execution time overhead.

Customization creates many different copies from an initial version of a program [7]. Each copy of the protected program is different in its binary shape, but it is functionally equivalent to other copies. Thus, published exploits to attack one version might not work with other customized versions. This kind of protection discourages diffusion of "cracks" but it does not aim at detecting and reacting to tampering.

Another way to inhibit cracks diffusion is pursued by means of techniques like *Software aging* [14] aiming at frequently distributing new updates of a program: this also allows dynamic renewal of software protection techniques embedded in the application.

As mentioned, our approach proposes a software-only solution. On the contrary, *trusted computing* initiatives [8] rely on a trusted hardware platform to build software authenticity from the ground up. For instance, an implementation of trusted computing principles was proposed using a trusted coprocessor and a modified Linux kernel [9]. In that work, the authors create a chain of trust where BIOS and coprocessor measure integrity of the operating system at startup, then the operating system measure integrity of applications, then applications measure integrity of dynamically loaded modules, and so on.

In conclusion, when compared to existing integrity checking techniques (both software-based and hardware-based), our approach extends prior art in several direction. First, it provides remote verification that checking has been actually performed. Furthermore, in our implementation the checking component can be replaced dynamically at run time by means of software updates. Such quality improves the overall strength of the technique we propose since attackers have limited time resources to break the checks (see Section 4 for further details).

3. TrustedFlow Principles and Prototype Architecture

The key element of our approach is the software module implementing the tag generator on board the client application. The module must conform to two basic principles.

- *Interlocking* describes the combination of the original application with the tag generator, so that they are bound to each other in an inseparable manner.
- *Hiding* describes countermeasures that are necessary for the tag generator to ensure that reverse engineering is practically infeasible.

On the client side, the Trusted Tag Generator (TTG) constantly generates an unpredictable flow of tags, constituting the continuous idiosyncratic signature (that cannot be forged) of the software's execution. Tags are attached to data sent by the client towards the server. Tags are bound to both the state of the TTG (e.g., current encryption key and number of tags previously sent) and traffic generated by client software. On the server side, the Trusted Tag Checker (TTC) entrusts the integrity of client software by verifying the correctness of tags flow.

For increased robustness, the algorithm used to generate the tags should not require a strong synchronization between TTC and TTG. For instance, current implementation uses a block cipher [10] in counter mode and includes the counter value among the data transmitted between the TTC and the TTG. Moreover, cryptographic functions can be employed to

bind tags to transmitted data. For example, a message authentication code (HMAC) of both the data unit and the related tag can be attached by the TTG to protect against alteration of data associated to a valid tag. Alternatively, a HMAC calculation can be part of the tag generation algorithm.

To show the applicability of our approach, we developed a prototype implementation of TrustedFlow relying on dynamic AOP that was deployed in a Java-based messaging service (i.e., a client/server chat system)

3.1. AOP tutorial

Aspect-Oriented Programming [11] is a new programming paradigm easing the modularization of crosscutting concerns in object-oriented software development. In particular, developers can remove scattered code related to crosscutting concerns from classes and placing them into elements called aspects. This methodology is implemented by different AOP platforms; all these tools rely on their own join-point model, which defines the points along the execution of a program that can be possibly addressed by an aspect.

Thus, AOP involves a compiling process, called weaving, for the actual insertion of aspect code into pre-existing application source code or bytecode. When using a dynamic AOP platform, e.g. PROSE [12], weaving can also occur at run-time.

In PROSE platform, an aspect is a normal Java class containing a set of *Crosscut* objects. A crosscut contains a method called *advice* and a *pointcutter* object identifying at which points in the dynamic execution of the program, advice code should be executed. For example, a pointcutter describes sets of join points by specifying the objects and methods to be considered, or a specific method execution.

PROSE offers a rich set of crosscuts: among these the 'MethodCall' crosscut intercepts method calls

PROSE uses a wildcard-based syntax to construct the pointcutters in order to capture join points that share common characteristics, and it provides logical operators to form complex matching rules by combining simple pointcutters.

Dynamic AOP platforms can be further distinguished in two categories: platforms with fixed pointcut definition and platforms with dynamic pointcut definition. In the former case the application code is instrumented once, at first deployment, in order to identify candidate join-points; then the related advice code can be added at load-time and then updated at runtime. PROSE use a dynamic pointcut redefinition: the application is not instrumented and the actual join-points are determined at runtime by the platform, depending on the pointcutters defined in the aspect.

We decide to use PROSE platform, because of its high dynamicity that can be an advantage for our purposes: in fact, if the attacker knew which parts of the application will be addressed by the aspects, (s)he could use them as a possible starting point for an attack.

3.2. Prototype design

We started from an existing prototype of chat system, previously developed in the context of a Java course. Thanks to the transparent use of dynamic AOP, the client program needs no changes. Therefore, once the chat client program is released and distributed to users, attackers cannot get clues about the integrity strategy the server will adopt. In contrast with a normal chat application, the client-side program must be executed within the PROSE runtime environment. Concerning the server program, it was extended to integrate with the TTC server module.

As shown in right-hand side of Figure 1, main components of the TrustedFlow prototype are the Aspect Manager, the Aspect Factory and the Code Checker. Aspect Manager provides the Chat Server with the interface to access TrustedFlow functionalities, namely the registration of a new client and the verification of tagged messages. Aspect Manager is assisted by the Tag Checker, which validates the tags carried by user messages, and by the Aspect Factory, which dynamically generates the code of the to-be-deployed aspect.

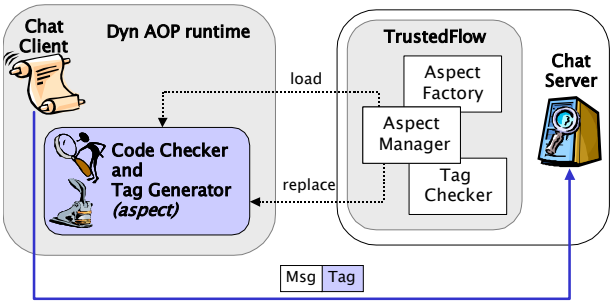

Figure 1. TrustedFlow with Dynamic AOP

As shown in left-hand side of Figure 1, when a new client comes in, a new aspect is crafted by the Aspect Factory and the Aspect Manager loads it in the execution environment of the upcoming client. The aspect implements both code integrity checker (that behaves as a watchdog for the client program and looks for integrity breaches) and the trusted tag generator aspect (that seamlessly appends a tag to each user message). Finally, note that the aspect can be replaced by the Aspect Manager at any moment during runtime.

4. Anti-Tampering with Dynamic Aspects

To counter reverse engineering, current software-

based tamper-resistance techniques rely on code checkers whose position is hidden in the application and whose behavior is obfuscated.

However, we observe that any technique involving a checker that is permanently embedded within the application is not robust enough. Indeed, the checker can be eventually identified and inhibited by an attacker with enough knowledge, time, and reverse engineering tools. Under such conditions, there are no guarantees that a remote client is actually undergoing to all the checks it is supposed to.

TrustedFlow overcomes such limitation by supporting dynamic replacement of the checker module, which is implemented as a dynamic aspect. The checker is bundled as an independent aspect, which is sent to the client at startup and can be updated dynamically at any time. The update rate can be tuned according to the security requirements of the target application domains (from minutes to days).

Nonetheless, other techniques, such as obfuscation, are still a valuable addendum in order to make it even harder for a rogue user to hack the checker code. A checker that is both mobile and obfuscated gives an additional degree of freedom to customize the security level: the stronger obfuscation, the lower the update rate can be, and vice versa.

Implementing anti-tampering techniques with AOP tools is intuitively useful because aspects can be seen as additional code having a privileged view on the application code and data. Moreover AOP weavers help "hooking" integrity checkers in the application code in a simple and flexible way, instead of using ad-hoc pre-compilers like most of the current approaches. In particular, the power of pointcutters composition rules is suitable for a flexible management and distribution of checking code in a large code base.

The dynamic aspect is composed by a tag generator crosscut, and two checkers crosscuts, i.e. the sandbox checker and the bytecode checker.

4.1 The Tag-Generator Crosscut

The Tag Generator crosscut intercepts network transmissions to the chat server in order to obliviously insert authentication tags in the data sent out by the client application.

The chat client relies on Java sockets to communicate with the chat server. The following code snippet of the Tag Generator shows the advice (method METHOD_ARGS) and the pointcutter which intercept all the messages sent by the client through a Java socket.

Each Tag Generator has a counter that is incremented each time a new message is sent out. The Tag Generator also shares a temporary symmetric encryption key with the TTC. Each client has a different key and keys can be changed at any time by updating the corresponding Tag Generator aspect.

```
1. public Crosscut tagGenerator = new MethodCut() {
2.   public void METHOD_ARGS(PrintWriter p, String msg) {
3.     StringBuffer tag = new StringBuffer(msg);
4.     tag.append( crypt( counter, key ));
5.     tag = hash(tag);
6.     p.println(tag);
7.     p.println(counter); p.flush();
8.     counter++;
9.  }
10. protected PointCutter pointCutter() {
11.   PointCutter socket =
12.     Within.method("println").AND(type("PrintWriter"));
13.   return Executions.before().AND(socket);
14. }
15.};
```

Figure 2. The Tag Generator Crosscut

Tags are generated according to the following algorithm:

- Using the secret key, the Tag Generator encrypts the current counter value with the AES block cipher (line 4).
- The resulting block is concatenated with the plain user message (line 4) and then hashed to produce the tag (line 5)
- The resulting tag and the current counter are pre-pended to the message network transmission (line 6).

Note that valid tags are generated unless the Tag Checker discovers a tampering attempt. In that case, the checkers invalidates the symmetric key used by the generator. On the receiver side, the server calculates the expected tag for the received message and compares the result with the tag sent by the client. In case of forged tags, the Chat Server reacts by inhibiting any further network communication coming from the suspected host; otherwise the plain message is relayed to clients.

The TrustedFlow module on the server side, automatically updates each Tag Generator whenever the aspect aging timer elapses.

Considering Figure 2, when a new Client signs in with the Chat Server (1), the latter informs the Aspect Manager (2) as to get the aspect pushed to the client environment. The Aspect Manager maintains a list of opened Client Sessions, each containing the TrustedFlow-related information for the corresponding Client, like the current shared key and the aging status of the deployed aspect.

Particularly, when a Client registers, a new Client Session instance is created (3). In turn, the Client Session contacts the Tag Checker singleton component to obtain a fresh key (4). Finally, the Aspect Factory generates the Java code for the customized versions of both Code Checker and Tag Generator aspects (5) and the Client session deploys them to the Client (6).

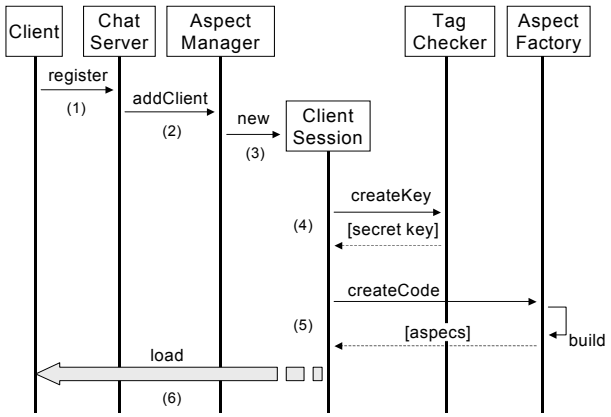

Figure 3. Client registration scenario

The aspect is automatically built by the Aspect Factory in the TrustedFlow module on the server side, using an Aspect Template that accommodates for the customization of main parameters, as the initial value of the counter and the secret key. Upon installation, each PROSE aspect can execute an initialization procedure before being woven. Our prototype exploits this feature to add extra security check. Namely, during the initialization phase, the newly deployed aspect verifies the checksum of older aspect bytecode, before withdrawing it. In case of mismatch, the newcomer aspect invalidates the encryption key.

4.2 The Code-Checker Crosscuts

Java's security model is focused on protecting users from hostile programs downloaded from un-trusted sources across a network. To accomplish this goal, Java provides a customizable "sandbox", which restricts the operations an application can perform.

```
1. public Crosscut sandbox = new MethodCut() {
2.  public void METHOD_ARGS(ANY x, REST pp) {
3.    key=null;
4.  }
5.  protected PointCutter pointCutter() {
6.    PointCutter native = Within.method(NATIVE_MODIFIER);
7.    PointCutter fork =
          Within.method("exec").AND(type("java.lang.Runtime"));
8.    PointCutter loader = Within.subType("java.lang.ClassLoader")).
9.          AND(NOT(Within.type("java.security.SecureClassLoader"));
10.   return Executions.before().AND(native.OR(fork).OR(loader));
11. }
12.};
```

Figure 4. The Sandbox Crosscut

In our case the problem is the opposite. The trusted aspects sent by the trusted server, in principle, cannot trust the environment they will be deployed in. In particular, an attacker could make use of the Java runtime to deceive the dynamic aspect. To this aim, deployed aspects prohibit many "dangerous" activities to the executing application. As shown in code fragment of figure 4, the following potentially insecure operations are disallowed: (1) call to native methods (line 6), (2) execution of external processes (line 7), and replacement of default secure class loader (line 8).

```
1.public Crosscut bytecodeChecker = new MethodCut() {
2.  public void METHOD_ARGS(ANY x, REST pp) {
3.    String className =
4.   thisJoinPoint().getThis().getClass().getName();
5.    String method =
6.   thisJoinPoint().getSignature().toLongString();
7.    if (!checkBytecode(className, method))
8.      key=null;
9.  }
10.  protected PointCutter pointCutter() {
11.   return Executions.before().
12.    AND(Within.packageTypes("it.polito.chat.*"));
13.  }
14.};
```

Figure 5. The Bytecode Checker Crosscut

Once the aspect is shielded against (naïve) disablement attacks, it can safely attend the checking task. To this aim, the aspect contains the bytecode checker crosscut (see Figure 5) resorts to BCEL Java library [14] to access the application bytecode and eventually calculate each method checksum (the above-mentioned encryption key is used). Checksums are then compared against an on-board list of pre-computed values. The advice (*METHOD_ARGS* method) is called whenever an application method is invoked. As shown on line 10, all calls within the application package ("it.polito.chat") are monitored. The aspect makes use of the PROSE API to extract the actual method signature and class name (lines 3 to 6). It then compares the bytecode hash with the original one (line 7): if they differ the key is nullified (line 8), and the tag generator will send wrong tags, implicitly notifying the server that something went wrong.

5. Threat analysis and discussion

The three main attacks to software integrity mechanisms are discovery, disablement, and replacement [3].

Discovery is the first step to disable or replace protections. Our approach makes discovery more difficult because the code-checking module is not bundled within the application deployed on the un-trusted host. Furthermore, even if the module is discovered at runtime, its limited time validity reduces attacker possibilities. Thus, static inspection tools are defeated by our approach, thanks to dynamic loading of the TTG module. Dynamic analysis tools, such as debuggers and profilers, pose a possible threat to our approach. However, the threat is moderate because the human is involved in the loop, and, through dynamic replacement, we bound the time available to the human in order to discovery the checker position/behavior.

The fact that AOP allows the integrity-checking code to be well-modularized and separated from the rest of the client could seem dangerous. However, the client-deployed aspect implements both the checking mechanism and the tag generation. Thus *disablement* attack is not a concern, as disablement of checker also disables the tag generator. Hence the server would detect the attack (tags are no longer generated).

As current dynamic AOP platforms do not take advantage of security technologies, our prototype is exposed to *replacement* attacks currently. That is, once the dynamic aspect is captured using packet sniffers, the attacker could decompile it, understand the TTG behavior, and replace it with a forged copy. It is unlikely that such a complex attack can be completed manually before the aspect expiration time elapses. Nonetheless, the only possibility to thwart an automated replacement attack is combining different protection techniques (e.g. obfuscation) and reducing the TTG time validity.

In general, a possible improvement to increase complexity of attacks would be based on an overlapping coverage of multiple integrity checking aspects, so that each aspect is validated by several others. The disablement of one or more of the aspect advices would be detected by the aspects still in place. Additionally, each aspect could perform different types of code integrity checking to improve the overall strength of our approach to the above-mentioned attacks. Finally, customization of aspect code would help either, so that inferring the behavior of a newly updated aspect starting from the previous one was difficult.

Finally, similarly to all software-based techniques, we rely on a possibly un-trusted external platform: the PROSE environment in our case. Reliance on such un-trusted support could make the whole system vulnerable. However, our prototype could be extended in order to check authenticity of underlying PROSE virtual machine as well.

6. Conclusions

We presented TrustedFlow, an innovative, all-in-software methodology to deal with remote verification of correct execution for client-side application code. The proposed solution extends state-of-the-art integrity-checking techniques by providing automated and periodic replacement of checking code during run-time. Furthermore, our approach supports the continuous attestation of integrity by a remote server.

We presented a prototype implementation of TrustedFlow that is based on Java and dynamic AOP. Aspect-oriented programming proved to be a powerful and effective technique to seamlessly weave the integrity checker with the application program. As a positive outcome, the strategy adopted by the checker is not visible through static code analysis, and the attacker duty is made even heavier thanks to continuous replacement of the checking aspect. In our prototype, integrity checking is based on secure checksums of executed bytecode, which are continuously compared with pre-calculated correct values.

7. References

[1] M. Baldi, Y. Ofek, and M. Yung, "Idiosyncratic Signatures for Authenticated Execution of Management Code" Proc. of DSOM 2003, 2003

[2] H. Chang and M. Atallah, "Protecting software code by guards" Proc. of ACM Workshop on Security and Privacy in Digital Rights Management, 2002

[3] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic Self-Checking Techniques for Improved Tamper Resistance" Proc. of ACM Workshop on Security and Privacy in Digital Rights Management, 2001

[4] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. Jakubowski, "Oblivious hashing: Silent Verification of Code Execution" Proc. of 5th International Workshop on Information Hiding (IHW 2002), 7-9 October, 2002

[5] C. Collberg, C. Thomborson, and D. Low, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection" *IEEE Transactions on Software Engineering*, 28, 2002.

[6] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (Im)possibility of Obfuscating Programs" Proc. of CRYPTO 2001, 2001

[7] D. Aucsmith, "Tamper resistant software: An implementation" in *Information Hiding, Lecture Notes in Computer Science 1174*, R. J. Anderson, Ed.: Springer-Verlag, 1996.

[8] TCG, The Trusted Computing Group, available at: https://www.trustedcomputinggroup.org (last access 30th May, 2005),

[9] R. Sailer, X. Zhang, T. Jaeger, and L. VanDoorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture" Proc. of 13th USENIX Security Symposium, 2004, pp. 223-238.

[10] J. Daemen and V. Rijmen, "The Block Cipher Rijndael" in *Smart Card Research and Applications, LNCS 1820*, Springer-Verlag, 2000.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwan. Aspect-oriented programming. Proc. of ECOOP 97, June 1997.

[12] A. Popovici, G. Alonso, and T. Gross, "Just in Time Aspects: Efficient Dynamic Weaving for Java" Proc. of 2nd International Conference on Aspect-Oriented Software Development, 2003

[13] BCEL, Byte Code Engineering Library, available at: http://jakarta.apache.org/bcel/

[14] M. Jakobsson, K. Reiter, "Discouraging Software Piracy Using Software Aging"**.** ACM Workshop on Security and Privacy in Digital Rights Management, Philadelphia, USA, November 2001.