IET Information Security

Research Article

IET Journals
The Institution of Engineering and Technology

Aspectising honeytokens to contain the insider threat

ISSN 1751-8709
Received on 7th February 2014
Accepted on 20th October 2014
doi: 10.1049/iet-ifs.2014.0063
www.ietdl.org

Keshnee Padayachee ✉

University of South Africa, UNISA, PO BOX 392, South Africa
✉ *E-mail: padayk@unisa.ac.za*

Abstract: The aim of this study is to provide a generic implementation strategy for honeytokens deployed within a database management system that leverages the aspect-oriented paradigm to contain the insider threat. This approach is tested by developing a proof-of-concept prototype in an aspect-oriented language, namely AspectJ. This study also reflects on design and implementation challenges involved in the deployment of honeytokens to contain the insider threat. Consequently, aspect-orientation is proposed as a means to resolve some of these challenges.

1 Introduction

The term 'insider threat' refers to any individual who works in an organisation and abuses the authority granted to him/her for illegitimate gain [1]. Examples of attacks include unauthorised extraction, tampering with data and deletion of critical assets [2]. Chinchani *et al.* [3] argue that since insiders have legitimate access, security administrators often consider this type of threat as inevitable. According to Spitzner [4], honeypots may be a way of containing the insider threat, that is, limiting the damage that is caused by deflecting the malicious insider to a honeypot or honeytokens. Although in theory the idea has merit, the practical considerations of developing such a system have not yet been fully explored within a database management system (DBMS). The first consideration involves the consequences of supplementing a legacy system with a non-functional concern, which will involve additional maintenance costs. Aspect-oriented programming (AOP) has long been advocated as a suitable methodology to augment a legacy system with supplemental functionality while reducing the maintenance overheads [5]. This paper aims to provide an insightful review of the use of AOP to implement honeytokens as a separable concern and it explores the general challenges involved in designing honeytokens.

A honeypot is an information system resource whose value lies in the unauthorised use of that resource [6]. Using honeypots should confirm insiders' actions and potentially analyse their motivations and resources [4]. Recent research has increasingly recommended honeypots to counter the insider threat [7, 8]. A honeypot is more than just a computer or a physical resource – it may be anything from a Windows program to an entire network of computers [6]. However, in its most rudimentary form, it may be a credit card number or a password. In this fundamental form, a honeypot is called a honeytoken [4]. The power of deploying honeytokens to counter the insider threat lies in their simplicity. Despite the fact that honeytokens are easily customisable and easily integrated [4], there are few practical examples that demonstrate how they may be integrated or optimised in an extant system. Typically, these types of security concerns are not considered in the original system. As problems become evident, systems are constantly upgraded with better protection mechanisms. Fortunately, AOP facilitates the integration of security concerns such as honeytokens into extant systems. Chew *et al.* [9] also advocate the use of aspect orientation to weave honeytokens into existing web applications, although they do not provide further details on how this would be accomplished.

AOP is strongly recommended for security-related concerns [10] such as authentication, access control and integrity [11]. Aspect-orientation has the potential to enhance the implementation of such cross-cutting concerns in terms of reusability and extensibility, thereby improving the robustness and maintainability of a system [12]. The objective in this paper is to seamlessly augment existing systems with honeytokens, with no perceptible perturbation to the extant system. It results in an implementation that is easier to maintain and port to different environments. Aspects improve the separation of concerns by making it possible to cleanly localise cross-cutting design concerns. They also allow programmers to write, view and edit a cross-cutting concern as a separate entity.

In this study, an effort is made to employ AOP to deploy honeytokens in DBMSs to contain the insider threat. This paper is structured as follows: Section 2 presents related work, whereas the design challenges involved in implementing honeytokens are discussed in Section 3. Section 4 contains a discussion of the concepts underlying AOP. In Section 5, an aspect-oriented prototype is demonstrated as a proof-of-concept. A case-in-point example is provided in Section 6. An evaluation of the prototype is presented in Section 7. Section 8 discusses the potential pitfalls of the model concept and this paper concludes in Section 9.

2 Related work

Mechanisms to counter the insider threat include monitoring, detection, mitigation [7] and deterrence. Monitoring alone is not sufficient to manage the insider threat. Since it captures the intent but not the motivation – it is difficult to identify patterns of misuse [13]. Deterrents alone cannot provide insight into the actual act of insider threat [14]. Intrusion detection mechanisms also present a number of challenges from false positives through to the effort required to correctly identify anomalous behaviour [7].

In view of the challenges associated with existing mechanisms to contain the insider threat, Spitzner [4] highlights several advantages to using honeypots as an alternative. Unlike intrusion detection systems, data are collected only when an insider interacts with a honeypot – making the data easier to analyse and manage. Using honeypots reduces false positives as any interaction with a honeypot is virtually guaranteed to be illicit. A limited number of applications have been developed to deploy honeytokens to combat the insider threat. The terminology in this field has not been fully established, as some researchers refer to honeytokens as decoys, honeyfiles or lures.

Yuill *et al.* [15] considered using honeyfiles to bait hackers. They argued that the implementation of honeyfiles was non-trivial – constructing deceptive files, file content and system footprints was

challenging as the process had to be consistent and believable. Bowen *et al.* [7, 16, 17] considered using decoy documents. Although they were able to test each component of deployment and the distribution of decoys, they were not able to integrate all the systems. They were also unable to develop a fully integrated detection system without a 'response component'. The current paper suggests aspect-orientation as a possible technique to resolve this problem, as it allows the seamless augmentation of supplemental functionality as a separable concern into an extant system.

Čenys *et al.* [18], who also integrated honeytokens into a DBMS, considered incorporating a honeytoken table, where any interaction with the table indicates malicious activity. They experienced challenges with implementing honeytokens, related to the complexities of developing triggers and external procedures for alerting security personnel, and stated that since a DBMS such as Oracle was more adequate in realising the design, the solution was not generic. However, the solution provided by Čenys *et al.* [18] does not describe how the honeytoken table would be made conspicuous to the insider threat. Honeypots are only of value when an attacker interacts with them and they capture only actions related to this activity [4]. Hence, this paper considers how honeytokens may be distributed so as to increase the odds of a malicious insider discovering them. Moreover, considering the micro-level of honeytoken records rather than the macro-level of a honeytoken table allows the reconnaissance process to be more fine-grained. The solution offered by Čenys *et al.* [18] lies at the database-level, whereas the solution presented in this paper is at the application-level. Integrating honeytokens at the database-level may prove difficult to maintain, as the procedures will be dependent on the database product. Furthermore, it does not localise all the concerns related to honeytokens, which results in easier maintenance and optimisations, and this is easily facilitated by a high-level expressive aspect-oriented language such as AspectJ. In the next section, the design concerns involved in automating honeytokens are explored.

3 Design concerns for implementing honeytokens

As a motivating example, consider a DBMS that maintains the voters' roll for an election. Suppose a malicious insider modifies the voters' roll to ensure that a particular candidate wins. Inserting honeytokens may help in detecting such an offence. If the insider unknowingly alters a honeytoken record as well, it is clear that such insider's intent is malicious, as this instruction would not have been authorised for a fictitious person.

According to Bowen *et al.* [16], decoys must display a number of properties in order to bait an insider. They must be believable, enticing and conspicuous; decoys must not be easily identifiable, and a decoy should not hinder, obstruct or impede normal operation. Gupta *et al.* [19] recommend that lure data must remain static until the confirmation/elimination of suspicion about a suspected user. They also suggest that lure data should be unique for each suspected user. Bowen *et al.* [16] in turn specify that a decoy should be detectable.

The properties identified by Bowen *et al.* [16] and Gupta *et al.* [19] were used to delineate four basic operations associated with honeytokens. First, there has to be a means of generating honeytokens that are believable, enticing, unidentifiable and specific to each insider. Second, honeytokens should function as normal data, so there has to be a management mechanism to update the honeytokens to allay suspicion. Third, a honeytoken should be detectable to security administrators, meaning that there has to be a means of detecting that a honeytoken has been 'attacked'. Fourth, a honeytoken has to be conspicuous, so there has to be an effective means of distributing the honeytokens.

3.1 Honeytoken generation

There are several design challenges that have to be taken into account when generating honeytokens. First, honeytokens are designed to mimic real data, hence there has to be means of distinguishing them from real data. Bowen *et al.* [7, 16, 17], for example, used embedded beacons that signal when the decoy document is opened. They concede that beacons may be used to snare the least sophisticated user, as the network connection of the beacon may be used as distinguishing feature and the beacon may fail. Bowen *et al.* [17] also tagged decoys with a cryptographic hash function. Although they believed that these would be undetectable to the average user, they conceded that the highly privileged user may be able to distinguish decoys from non-decoys. To create true 'indistinguishability' between decoys and non-decoys, Bowen *et al.* [17] tagged both decoys and non-decoys. Such a technique may however prove to be cumbersome in a database context, since integrating a tag in all records to identify a few malicious insiders may result in system performance issues. Furthermore, as discussed earlier, implementing this technique in an extant system could prove difficult and it may be prudent to store honeytokens in a separate DBMS so as not to compromise the integrity of an organisation's genuine data.

The second challenge involves generating honeytokens that are believable and enticing. White [20] and Bercovitch *et al.* [21] attempted to resolve this problem by using real data as a basis. Yet, a honeytoken is useless if it is not conspicuous enough to be selected by a malicious insider. This paper recommends seeding the database with honeytoken records as described by either White [20] or Bercovitch *et al.* [21] and updating the honeytokens to match the queries that the insider executes. For example, consider an insider who queries a table named VoterRoll table for all voters who live in 'Menlyn', as shown in Fig. 1. The honeytoken generator will re-engineer a number of honeytoken records to match that query by changing the city to 'Menlyn'. This ensures that the insider will be enticed by the honeytoken.

The next issue under consideration involves the distribution of the honeytokens generated.

3.2 Honeytoken distribution

As it is not practical to distribute all the honeytokens instantaneously, a mode has to be associated with each honeytoken record. For instance, some honeytoken records should be set with a mode of being available for selection and others will be set on reserve. In case some honeytoken records have been tampered with, they should be updated just as the insider expects, otherwise this may arouse suspicion. When some honeytokens have been 'deleted' by the insider (of course they are not actually deleted), they should not appear in subsequent searches and should be marked with a closed mode.

3.3 Honeytoken management

Honeytokens should react as normal data – hence they should not hinder, obstruct or impede normal operations. When an insider

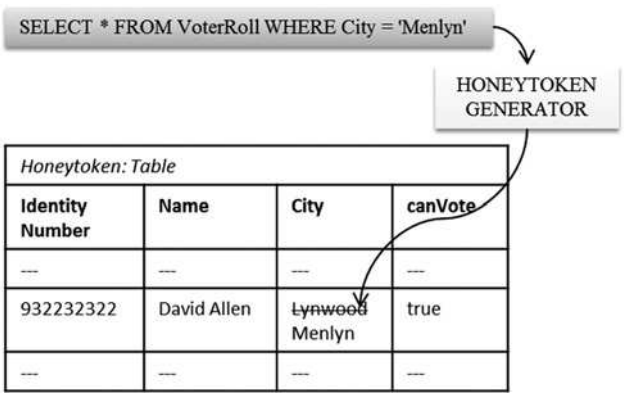

Fig. 1 *Query-based honeytokens*

attempts an action on the honeytoken, the action must be carried out as the insider presumes. For example, if the insider updates the honeytoken, the management process must perform this update on the honeytoken table without interrupting the flow of processes. The insider should be allowed to perform the command without discernable difference from the typical functioning of the system. The management process addresses a salient property of honeytokens, that is, they must remain static until confirmation/elimination of suspicion about a suspected insider has been ratified. The honeytoken manager needs to re-engineer the query to be applicable to the honeytoken, as the insider assumes it exists within the actual database. The honeytoken manager conducts these operations automatically without human intervention.

3.4 Honeytoken detection

This exercise involves detecting when the user is performing some operation related to the honeytoken record, such as updating or deleting it. The purpose of detection is to initiate a reconnaissance mission and to invoke methods to handle the management of the update as was described previously.

Although it is sensible to deploy honeytokens in a DBMS, the problem remains their implementation. It is challenging to evolve an extant system to contain honeytokens. This type of upgrade will subject an extant DBMS to numerous regression errors. Hence, it is sensible to implement the processes of honeytoken deployment each as a separable concern. This facilitates development of the honeytoken processes by an external task team that has no association with the organisation and so ensures the security of the system and the anonymity of the honeytokens.

4 Aspect-oriented programming

AOP provides explicit language support for modularising design decisions that cross-cut a functionally decomposed programme [22], which allows the developer to maintain the code (cross-cutting functionality) in a modularised form. It is important to note that AOP maintains all the benefits of the object-oriented programming (OOP) paradigm and it should be viewed as an extension rather than a replacement of object-oriented technologies.

The key concepts of AOP as described next are derived from Kiczales *et al.* [23]. AOP has a construct similar to classes in OOP, called aspects. An aspect consists of constructs termed pointcuts and advices. The pointcut designators are a set of join points described by a pointcut expression. Specifically, join points refer to well-defined points in a programme's execution such as when a method is called, when an exception is thrown etc. An advice defines the functionality to be executed before (i.e. the 'before' advice) or after (i.e. the 'after' advice) a defined join point or even possibly instead of the defined join point (i.e. the 'around' advice). An aspect weaver is then used to merge extant code (i.e. the base code) and aspects into a final programme.

The abstraction offered by the AOP assists in separating the roles of application developers and security specialists [11]. AOP is also flexible in that it supports wildcards, so there is no need for explicit naming [24]. As security evolves, AOP makes it easier to 'swap in and out and evaluate alternative treatment options' [25]. AOP has the potential to reduce the complexity of programmes and to improve the maintainability of software because of the additional level of abstraction offered. The requirements for a honeytoken deployment would be similar in any DBMS; hence, the implementation strategy presented in the next section is highly extensible and reusable.

5 Aspectising honeytokens

The aspect-oriented architectural design (Fig. 2) is based on the recommendations presented in Section 3 and an overview of the core operations of the aspect is shown in Fig. 3.

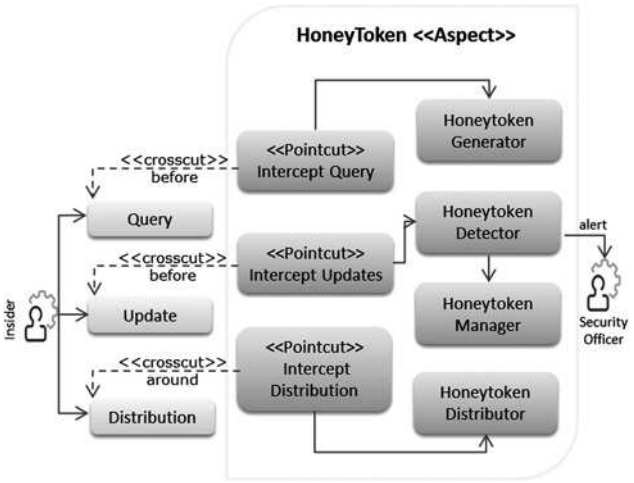

Fig. 2 *Aspect-oriented honeytoken deployment architecture*

5.1 Aspect-oriented honeytoken deployment architecture

It is clear that the aspect will need to satisfy four focus areas, namely, generation, management, detection and distribution. Where these operations should be invoked, depends on specific points in the execution. As the generation of honeytokens is based on queries, the aspect needs to cross-cut all join points 'before' the user 'executes a query' by activating a module to generate honeytokens (i.e. the honeytoken Generator) proactively. The detection and management process occurs when an 'update' is performed on a honeytoken. Therefore the aspect cross-cuts 'before' the join point where the insider executes an update. The module that handles the detection (i.e. the honeytoken detector) simultaneously triggers an alert (for instance to a security administrator) and activates the honeytoken manager. The honeytoken manager module processes

```
public aspect HoneyTokenAspect{
    ...
    HoneyTokenAspect() throws Exception{...} //constructor
    //core methods
    public void honeytokenGenerator(...){...}
    public boolean honeyTokenDetector(...){...}
    public void honeyTokenDistributor(...) {...}
    public int honeyTokenManager(...){...}

    //pointcuts
    pointcut interceptQuery (....): call (* executeQuery(...)) ...;
    before (...): interceptQuery (query) {...}

    pointcut interceptDistribution(...): call (*  Distributor(...)) ...;
    Object around (...): interceptDistribution(....) {...}

    pointcut interceptUpdate (...):call (* executeUpdate (...))...;
    before (...): interceptUpdate (...) {...}

    //Auxillary Methods
    public void honeyTokenSeed(...){...}
    public  String reEngineerSelectionQuery(...){...}
    public String reEngineerQuery4HoneyTokenTable(...){...}
    void alert(....){...}
}
```

Fig. 3 *Outline of the honeytoken aspect*

the updates on honeytokens so that they appear executed exactly as the insider would have predicted and therefore do not arouse suspicion. As the honeytokens need to be interwoven with real data, an alternative method for distributing real data with honeytokens needs to be invoked. This alternative distribution method (i.e. the honeytoken distributor) cross-cuts the current distribution or output module, and therefore operates 'around' the extant distribution method. The honeytoken distributor is concerned with the output of honeytokens interwoven with real data.

The pointcut InterceptQuery identifies those execution points where the insider performs a 'query' and activates the honeytoken generator. The pointcut InterceptUpdate identifies those execution points where the insider performs an 'update' and activates the honeytoken detector and honeytoken manager modules. The InterceptDistribution pointcut identifies those execution points where the insider performs an output functionally and activates the honeytoken distributor module.

5.2 Proof-of-concept

The architectural design described was codified into an aspect (Fig. 3). The HoneyTokenAspect comprises the core methods and the pointcuts, as well as a few auxiliary methods (see Appendix for details, Fig. 3).

For the sake of simplicity, it is assumed that there is only one table of real data in the DBMS. The constructor in HoneyTokenAspect is responsible for creating a database that is nearly identical to the real database, except for the fact that the honeytokens table (i.e. HoneyTokens) contains two additional fields – mode and userName – to hold the username of an insider who tampers with the record. The mode is set to open, closed, tampered or reserved. honeytokens will be set to closed mode if the insider deletes them. The tokens set to closed will not be visible.

5.2.1 Overview of the core modules: The honeyTokenGenerator method generates the honeytokens for the HoneyTokens table. The honeyTokenGenerator method invokes the auxiliary method honeyTokenSeed to seed the HoneyTokens table with numerous representative records. To create conspicuous honeytokens, the honeytokens are refined using user-based queries. The method reEngineerSelectionQuery re-engineers the selection query into an update query on the HoneyTokens table, thereby ensuring that the insider is supplied with tokens that match his/her query. The method reEngineerSelectionQuery re-engineers a query such as 'SELECT * FROM VoterRoll where voterCity = 'Menlyn'' into 'update HoneyTokens set voterCity = 'Menlyn' WHERE Status = 'OPEN'' using string manipulation. The honeyTokenGenerator should therefore use real queries and generate false data that appear similar in nature by re-engineering a few existing honeytokens to match (i.e. convert the selection query into an update query on the honeytoken database). The number of honeytokens that should be generated is context dependent. It is assumed that larger data sets would require more tokens to be generated in order to increase the conspicuousness of the honeytokens. A honeytoken cannot be re-engineered twice as it contravenes the property that lure data must remain static.

The honeyTokenManager method allows updates on honeytokens to appear to be executed exactly as the insider presumes. As the insider assumes that the update has been performed on real data, the query has to be adapted to update the HoneyTokens table instead. The reEngineerQuery4HoneyTokenTable method is responsible for re-engineering the updates accordingly.

The honeyTokenDetector method invokes the honeyTokenManager and if the latter then performs an update, this implies that a honeytoken has been tampered with. The honeyTokenDistributor method manages the output by weaving in honeytokens among the real data.

5.2.2 Overview of the interceptor pointcuts: The first pointcut interceptQuery intercepts database queries and invokes the honeyTokenGenerator to generate tokens similar to the query. Simply put this pointcut intercepts any call to the method executeQuery and captures the augments at the point. The augments are required in the advice to process the query. This advice is executed 'before' the executeQuery method in the extant base code executes. An executeQuery method is called to perform an SQL query. Typically, a structured query language (SQL) SELECT statement is executed, where records matching the search condition in the WHERE clause are returned [26]. The output from a SELECT statement is typically displayed on the screen; hence, it is clear that a display type method to distribute this data would be invoked next. If this occurs, the aspect will need to interweave tokens with the real data. To prepare for this possibility, the aspect will retain a table of data that represents the database result set of the query on the honeytokens table as a global variable.

The second pointcut interceptDistribution intercepts the method that distributes (i.e. outputs) genuine data in order to interweave honeytokens with the real data. The method honeyTokenDistributor merges real data with those honeytokens, which are available (i.e. those in open mode). The advice advises 'around' the join point and invokes the honeyTokenDistributor method.

The third pointcut interceptUpdate detects updates on honeytokens by intercepting calls to executeUpdate. The method executeUpdate executes SQL statements to update a table [26]. This advice is executed 'before' the executeUpdate method in the base code executes. The method honeyTokenDetector determines whether a query contains a honeytoken; it then invokes the honeyTokenManager, which will update the HoneyTokens table if the query is related to a honeytoken. If the HoneyTokens table was successfully updated, then the honeyTokenDectector method returns true and the alert method which is intended for surveillance purposes is invoked. If the executeUpdate method invoked in the base code does not contain a honeytoken, the extant code is allowed to proceed as normal.

The aspect is fairly generic and reusable. The current prototype demonstrates that honeytoken deployment may be treated as a separable concern. To illustrate the inner workings of the aspect, a worked example is presented next.

Table 1 Extract from VoterRoll table

VoterRoll: table			
Identity number	Name	City	canVote
7 312 099 981	Paul Smith	Menlyn	true
7 505 062 929	David Alan	Menlyn	false
7 605 062 929	Cindy Shaw	Sunnyside	false
7 705 062 929	Mandy Ryan	Menlyn	false

6 Worked example

Consider a database that maintains a voters' roll as described in Section 3. The database has one table named VoterRoll. The VoterRoll table keeps a record of each voter with the following attributes: Identity Number, Name, City and canVote (i.e. whether the voter is eligible to vote). Suppose VoterRoll has the following subset of records as shown in Table 1.

Consider the following scenario: if a malicious insider anticipates that a certain candidate will win the election in a particular area, he may decide to tamper with the electorate details in this area. He

Table 2 Extract from HoneyTokens table

HoneyTokens: Table				
Identity number	Name	City	CanVote	Mode
7 677 744 455	Alice David	Sunnyside	true	OPEN
7 667 389 999	Sam Well	Brooklyn	false	OPEN

begins by querying the database. Suppose the city of concern is 'Menlyn'. To avoid suspicion, he randomly changes (or deletes) the records of voters who are entitled to vote to a status where they are no longer eligible to vote. Assume the HoneyTokens table has been seeded with the following subset of honeytokens as shown in Table 2.

Suppose that the following lines of code are executed in the extant system as shown in Fig. 4.

The statement at point A implies that the user would like to select all records from table VoterRoll where the voterCity is Menlyn. The pointcut interceptQuery advises all executeQuery statements 'before' they are executed, as shown symbolically on the left-hand side of the code. The pointcut advises this join point by calling the honeyTokenGenerator to generate a few tokens to match this query. Hence, the voterCity attribute in the subset of records in the honeytoken table will be reset to Menlyn.

The pointcut interceptDistribution is based on the premise that there will be a method to distribute the records in the base code after a query. This pointcut will 'wrap' around the distributor method (indicated at point **B**). The advice of the pointcut invokes the honeyTokenDistributor method. The distributor method in the base code does not execute; instead, the method honeyTokenDistributor is executed. This displays the honeytokens interwoven with the real data as shown in Table 3.

Now suppose the insider attempts to execute the following updates indicated at points **C** and **D**. The statement at point **C** attempts to reset the canVote field of a honeytoken record to false. The statement at point **D** attempts to delete a honeytoken record. The interceptUpdate pointcut detects this join point, as it contains a call to executeUpdate. The advice of interceptUpdate is executed 'before' this statement is executed. It determines whether any honeytokens are involved in the updates by invoking the honeyTokenDetector, which in turn invokes the honeyTokenManager. The honeyTokenManager revises the query to a query applicable to the honeytokens table. If the query executes successfully, then it implies that a honeytoken was involved, and the interceptUpdate advice immediately invokes the alert method. The statements at points **E** and **F** are invoked to display the updated table. (NB: no new honeytokens are generated as the query is generic.) The distributor method is intercepted by the interceptDistribution pointcut and the executeQuery is intercepted by interceptQuery pointcuts. The output will appear as shown in Table 4.

The honeytokenDistributor does not display the record that has been deleted, although the record still exists in the HoneyTokens table while the 'Alice David' honeytoken record has been updated to false as the insider expects. The worked example proves that the aspect consistently intercepts calls to query, update or display the database and that it deploys honeytoken processes effectively.

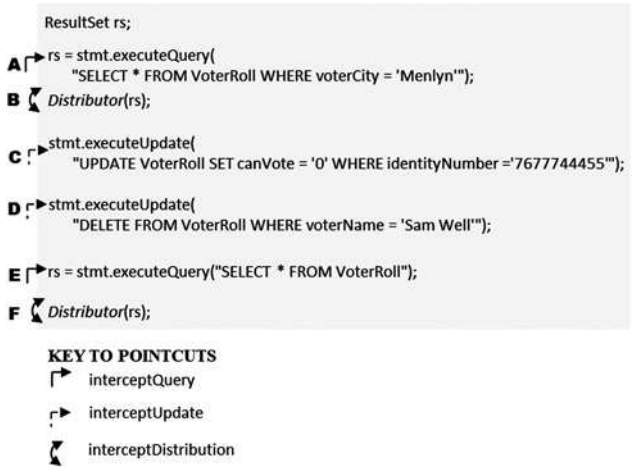

Fig. 4 *Extract of base code advised by pointcuts*

Table 3 Honeytokens distributed with records from table VoterRoll

Identity number	Name	City	CanVote
7 312 099 981	Paul Smith	Menlyn	true
7 505 062 929	David Alan	Menlyn	false
7 667 389 999	Sam Well	Menlyn	false
7 677 744 455	Alice David	Menlyn	true
7 705 062 929	Mandy Ryan	Menlyn	false

7 System evaluation

The empirical evaluation considers system performance issues such as memory usage and response time by conducting several experiments. Section 7.1 involved stress testing the prototype with large volumes of honeytokens. Section 7.2 involved comparing the OOP version of the prototype with the AOP version of the system concept.

7.1 Experiment 1

The memory usage is stable for honeytokens ranging from 100 to 1000. The honeytokens consume $\sim$2.5 times more memory than a system without any honeytokens (see Fig. 5*a*).

The speed of the performance system was measured for tokens from 100 to 1000. When there were no tokens, the system performed more than twice as fast. However, the system speed was not dramatically compromised with each increase of honeytokens – the reduction in response time occurred according to a gradual linear progression (see Fig. 5*b*).

7.2 Experiment 2

In this experiment, the aspect was converted into a class by transforming the advices into methods. Table 5 summarises the execution times based on three scenarios – each scenario is based on the number of honeytokens, as shown in Table 5. It can be noted that the object-oriented version is faster than the aspect-oriented version.

The memory footprints of each approach were measured under the same three scenarios as described earlier. Table 6 summarises the memory footprint for each scenario and draws a comparison between their memory usages. On average, the object-oriented version was found to use half as much memory as the AOP version.

AOP is clearly not preferable to OOP regarding system performance; the benefits are rather related to modularity. In this experiment, the code related to honeytoken deployment cross-cut the code base in seven instances. However in a real-world system the code scattering would be significant. The limited complexity of the system may restrict the extrapolation of the results of the experiments.

8 Discussion

It is evident that most security mechanisms are subject to 'diminishing efficacy', which means that over time they may lose their mitigating effect. It is proposed that the technique presented here should be upgraded regularly with variation. If an aspect-orientated approach is employed, these variations may be

Table 4 Extract of honeytokens distributed with records from table VoterRoll

Identity number	Name	City	CanVote
7 312 099 981	Paul Smith	Menlyn	true
7 505 062 929	David Alan	Menlyn	false
7 605 062 929	Cindy Shaw	Sunnyside	false
7 677 744 455	Alice David	Menlyn	false
7 705 062 929	Mandy Ryan	Menlyn	false

IET Inf. Secur., 2015, Vol. 9, Iss. 4, pp. 240–247

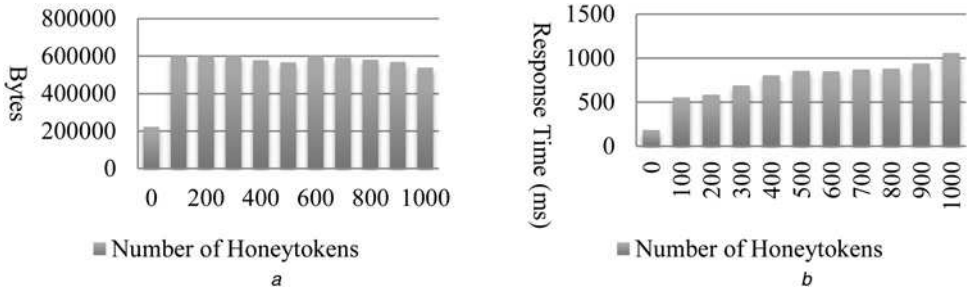

Fig. 5 *Results of Section 7.1*

a Memory footprint
b Response time

swapped in and out with minimal perturbation to the system. Honeytokens may be rendered ineffective if the insider is aware of their purpose and implementation. The AOP approach allows a higher level of abstraction where developers of security concerns are able to implement the honeytoken functionality independently. In this way, the code is protected from insiders who may use this type of knowledge to subvert the system.

The model concept proposed is partially subject to 'false positives'. However, considering the following scenario, an insider is presented with real data and honeytokens as prescribed by the model concept. If the insider performs a global update on both the real data and the honeytokens, it is possible that this act was unintentional. However, if the insider updates only specific honeytokens, then it is certainly probable that this insider's intent was malicious as such an update would not have been instructed on fake data. It is clear that insiders' maliciousness is a function of both their direct manipulation of the honeytoken and their commands to the database. Additionally, since an innocent user is likely to manipulate a honeytoken by accident fewer times than would a malicious user, there will be a clear pattern of abuse. Contextual factors that surround the attack of the honeytoken need to be considered before red-flagging an insider.

It can be argued that honeytoken deployment may affect productivity. However, when an AOP approach is adopted, the honeytoken strategy may easily be turned off during peak periods with minimal perturbation. Another possibility is to apply restrictions to honeytoken generation, which involves maintaining a set of properties when it should not react. In this context, whitelisting a set of authorised commands could be used to ensure that the honeytoken generator does not react in every circumstance. Certain individuals with a higher security clearance could, for example, also be exempted from being subjected to honeytokens. Furthermore, honeytoken deployment necessitates a clear separation of duties so that it does not compromise work-related performance. For example, consider the roles of reporting and updating data. Honeytoken deployment within the reporting role is inadvisable as it will result in inconsistencies. Although honeytoken deployment within the updating role is unlikely to compromise the efficacy of a benign insider. Hence, it is imperative these two roles are not assigned to the same individual.

The data of an organisation is a highly valued asset – any form of misuse of the data can irrevocably damage the reputation of an organisation. Organisations need to weigh the costs of deploying honeytokens (e.g. the increased performance overhead) against its potential benefits.

Table 5 Execution time comparison

Number of honeytokens	Execution time, ms		AOP version slower by, %
	OOP	AOP	
10	296	360	22
100	375	438	17
1000	906	1048	16

Table 6 Memory footprint comparison

Number of honeytokens	Object-orientation, kB	Aspect-orientation, kB	Percentage increase in memory footprint, %
10	262.93	587.00	123
100	262.94	587.01	123
1000	229.24	525.61	129

9 Conclusion

In this study, AOP was used to seamlessly augment an extant DBMS with the basic processes associated with honeytoken deployment. To demonstrate the validity of the design, a prototype was implemented and evaluated. AOP and the introduction of honeytokens were found to involve a substantial overhead; however, the benefits in terms of modularity are significant. Furthermore, research should involve evaluating the prototype by conducting usability tests with end-users.

10 References

1 Schultz, E.E.: 'A framework for understanding and predicting insider attacks', *Comput. Secur.*, 2002, **21**, (6), pp. 526–531
2 Salem, M.B., Hershkop, S., Stolfo, S.J.: 'A survey of insider attack detection research: beyond the hacker', in Stolfo, S.J., Bellovin, S.M., Keromytis, A.D., Hershkop, S., Smith, W.S., Sinclair, S.: (Eds.) 'A survey of insider attack detection research: beyond the hacker' (Springer, New York, 2008), pp. 69–90
3 Chinchani, R., Iyer, A., Ngo, H.Q., Upadhyaya, S.: 'Towards a theory of insider threat assessment'. Int. Conf. on Dependable Systems and Networks, Los Alamitos, Yokohama, Japan, June–July 2005, pp. 108–117
4 Spitzner, L.: 'Honeypots: catching the insider threat'. 19th Annual Computer Security Applications Conf. (ACSAC 2003), Los Alamitos, Las Vegas, USA, December 2003, pp. 170–179
5 Laney, C.R., Van der Linden, J., Thomas, P.: 'Evolution of aspects for legacy system security'. AOSD Workshop on Dynamic Aspects AOSDSEC'04, Lancaster, UK, March 2004, pp. 1–7
6 Spitzner, L.. Available at http://www.securityfocus.com/infocus/1713, accessed March 2012
7 Bowen, B.M., Salem, M.B., Hershkop, S., Keromytis, A.D., Stolfo, S.J.: 'Designing host and network sensors to mitigate the insider threat', *IEEE Secur. Priv.*, 2009, **7**, (6), pp. 22–29
8 White, J., Panda, B.: 'Implementing Pii honeytokens to mitigate against the threat of malicious insiders'. IEEE Int. Conf. on Intelligence and Security Informatics (ISI'09), Piscataway, Dallas, Texas, 2009, p. 233
9 Chew, B., Wang, J., Ma, A., Bigham, J.: 'Anomalous usage in web applications'. Networking and Electronic Commerce Research Conf. (NAEC'08), Lake Garda, Italy, September 2008
10 Viega, J., Evans, D.: 'Separation of concerns for security'. ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, Limerick, Ireland, 10 June 2000, pp. 125–138
11 Bodkin, R.: 'Enterprise security aspects'. Proc. of the AOSD Technology for Application-Level Security Workshop, Lancaster, UK, March 2004, pp. 1–12
12 Padayachee, K., Eloff, J.H.P.: 'The next challenge: aspect-oriented programming'. Proc. of the Sixth IASTED Int. Conf. on Modelling, Simulation and Optimization, Anaheim, Gaborone, Botswana, September 2006, pp. 304–307

13 Hunker, J., Probst, C.W.: 'Insiders and insider threats – an overview of definitions and mitigation techniques', *J. Wirel. Mobile Netw. Ubiquit. Comput. Dependable Appl.*, 2011, **2**, (1), pp. 4–27
14 Willison, R.: 'Understanding the perpetration of employee computer crime in the organisational context', *Inf. Organ.*, 2006, **16**, (4), pp. 304–324
15 Yuill, J., Zappe, M., Denning, D., Feer, F.: 'Honeyfiles: deceptive files for intrusion detection'. Proc. from the Fifth Annual IEEE SMC Information Assurance Workshop: Workshop Papers, Piscataway, United States Military Academy, West Point, New York, June 2004, pp. 116–122
16 Bowen, B.M., Salem, M.B., Hershkop, S., Keromytis, A.D., Stolfo, S.J.: 'Baiting inside attackers using decoy documents'. Security and Privacy in Communication Networks: Fifth Int. ICST Conf. (Secure Comm 2009), Berlin, Athens, Greece, September 2009, pp. 51–70
17 Bowen, B.M., Salem, M.B., Keromytis, A.D., Stolfo, S.J.: 'Monitoring technologies for mitigating insider threats', in Probst, C.W., Hunker, J., Gollmann, D., Bishop, M.: (Eds.) 'Monitoring technologies for mitigating insider threats' (Springer, New York, 2010), pp. 197–217
18 Čenys, A., Rainys, D., Radvilavičius, L., Goranin, N.: 'Implementation of honeytoken module in DBMS Oracle 9ir2 enterprise edition for internal malicious activity detection'. IEEE Computer Society's TC on Security and Privacy, Los Alamitos, Oakland, California, May 2005, pp. 1–13
19 Gupta, S.K., Damor, R.G., Gupta, A., Goyal, V.: 'Luring: a framework to induce a suspected user into a context honeypot'. IEEE Second Int. Workshop on Digital Forensics and Incident Analysis (WDFIA 2007), Los Alamitos, Samos, Greece, August 2007, pp. 55–64
20 White, J.: 'Creating personally identifiable honeytokens'. Innovations and Advances in Computer Sciences and Engineering, Springer, Netherlands, 2010, pp. 227–232
21 Bercovitch, M., Renford, M., Hasson, L., *et al.*: 'Honeygen: an automated honeytokens generator'. IEEE Int. Conf. on Intelligence and Security Informatics (ISI), Piscataway, Beijing, China, July 2011, pp. 131–136
22 Walker, R.J., Baniassad, E.L.A., Murphy, G.C.: 'An initial assessment of aspect-oriented programming'. Proc. of the 21st Int. Conf. on Software Engineering, New York, Los Angeles, CA, 16–22 May 1999, pp. 120–130
23 Kiczales, G., Hillsdale, E., Hugunin, J., *et al.*: 'Getting started with AspectJ', *Commun. ACM*, 2001, **44**, (10), pp. 59–65
24 Kiczales, G., Hugunin, J., Kersten, M., *et al.*: 'Semantics-based crosscutting in AspectJ'. Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, Limerick, Ireland, June 2000, pp. 1–6
25 Houmb, S.H., Georg, G., France, R., Matheson, D.: 'Using aspects to manage security risks in risk-driven development'. Third Int. Workshop on Critical Systems Development with UML, Lisbon, Portugal, October 2004, pp. 71–84
26 Shelly, G.B., Cashman, T.J., Starks, J.L., Mick, M.L.: 'Java programming comprehensive concepts and techniques' (Thomson Course Technology, Boston, 2006, 3rd edn.)

11 Appendix

See Fig. 6.

Appendix A

```
public void honeyTokenGenerator(String query) throws Exception{
    connection = DriverManager.getConnection(url);
    statement = connection.createStatement();
    honeyTokenSeed(connection,statement);
    String updateQuery = reEngineerSelectionQuery(query);
    if (updateQuery!=null) {
        statement.executeUpdate(updateQuery);
    }
    connection.close();
}
```

```
public int honeyTokenManager(String query) throws Exception{
    int numberOfUpdates = 0; //indicates the number of records updated
    if ((!query.contains("DROP")) && (!query.contains("CREATE"))){
        connection = DriverManager.getConnection(url);
        statement = connection.createStatement();
        String updateQuery = reEngineerQuery4HoneyTokenTable(query);
        numberOfUpdates = statement.executeUpdate(updateQuery);
        connection.close();
    }
    return numberOfUpdates;
}
```

```
public boolean honeyTokenDetector(String query){
    try {
        if (honeyTokenManager(query) > 0){
        return true;
        }
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return false;
}
```

```
pointcut interceptQuery (String query):
!within(HoneyTokenAspect+) && call (* executeQuery(String)) && args(query);
before (String query) : interceptQuery (query){
        String updateQuery;
        try {
            honeyTokenGenerator(query);
            connection = DriverManager.getConnection(url);
            statement = connection.createStatement();
            updateQuery = reEngineerQuery4HoneyTokenTable(query);
            rsHoneyToken = statement.executeQuery(updateQuery);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
pointcut interceptDistribution(ResultSetrsTrueData):
!within(HoneyTokenAspect+)  && call (* Distributor(ResultSet))
&& args (rsTrueData);
Object around (ResultSet rsTrueData): interceptDistribution(rsTrueData) {
```

```
        try {
            honeyTokenDistributor(rsTrueData,rsHoneyToken);
        }
        catch (Exception e) {
            e.printStackTrace();
    }
    return 0;
}
```

```
pointcut interceptUpdate (String query)  :
!within(HoneyTokenAspect) && call (* executeUpdate (String)) && args(query);
before (String query): interceptUpdate (query) {
        if (honeyTokenDetector(query)){
            try {
                    alert(query);
            }
            catch (Exception e)
                    {e.printStackTrace();
            }
    }
}
```

Fig. 6 *Appendix figure*