A SPL Framework for Adaptive Deception-based Defense

Cristiano De Faveri
NOVA LINCS, Department of Computer Science
Faculty of Science and Technology
Universidade NOVA de Lisboa
c.faveri@campus.fct.unl.pt

Ana Moreira
NOVA LINCS, Department of Computer Science
Faculty of Science and Technology
Universidade NOVA de Lisboa
amm@fct.unl.pt

Abstract

In cyber defense, integrated deception mechanisms have been proposed as part of the system operation to enhance security by planting fake resources. The objective is to entice attackers and confuse them in determining the legitimacy of those resources. Although several strategies exist to implement deception in a software system, developing and integrating such solutions are primarily made in an ad-hoc fashion. This hinders reuse and does not consider the operation life cycle management. Additionally, support for adaptive deception is not considered. To alleviate these problems, we propose a framework based on software product lines and aspect-oriented techniques to generate adaptive deception-based defense strategies. We illustrate the feasibility of our approach with an example from the web applications domain, by integrating honeywords into an authentication mechanism to mitigate offline password cracking attacks.

1. Introduction

Modern software systems intrinsically operate over a complex, hyper-connected eco-system comprising a diverse range of third-party components, accentuating the systems exposure to adversaries. Attackers constantly evolve their tactics to bypass security mechanisms and exploit zero-day vulnerabilities. This calls for more advanced security techniques to actively engage adversaries with customized responses and mitigation strategies. Studies on evolutionary biology show that deception plays a key role in the evolution of creatures, representing an advantage over their competitors [1]. This idea has echoed in the virtual world where the use of deception techniques for defense is currently a relevant component of active defense [2]. Deception aims at altering the perception of attackers by using negative information (fictions) or deliberate actions in favor of the defense [3].

Perhaps the most representative research area on deception-based defense lies on honeypots [4]. A honeypot is classified as a security resource designed to entice attackers; its value lies in being compromised. Traditionally, honeypots are built as confined network endpoints, i.e., sandboxed systems (physical or virtual) configured to appear vulnerable to adversaries but without any integration with real systems. However, the threat landscape is very dynamic, and deception mechanisms have evolved to be part of the system operations, constituting multi-layer deception-based defense [5]. Examples of integrated deception defense include deceptive password cracking and leakage [6, 7], software diversity [8], and, more recently, integrated honeypot operation [9].

Several strategies exist to implement deception in a system [2]. A program or even the operating system can be modified by hand to incorporate deceptive elements. More systematically, deception is integrated by using wrappers [10] on regular code and generalized by deception control policies that govern the level of suspiciousness, resources in the operating system/application, actions applied to resources, and the deceptions that should be applied. Coarse-grained approaches (e.g., honeypots and honeynets) use virtualization technologies to simulate real systems, requiring a significant amount of configuration and tuning to look like a real system [11]. Despite the advances in techniques to implement these strategies, there is a lack of systematic approaches that manage integrated deception life cycle, especially at the application level. Such an approach is important because the implementation of deception is not a trivial task [12]. Some situations require the operation to adapt to events occurring in the system context in aim of maintaining a plausible story. From the software engineering perspective, implementing deception and decoys adds to the concerns developers must deal with. Thus, reuse and seamless integration of deception into system components are key aspects towards a wider adoption of such technologies.

This paper proposes an approach to enhance the development process of self-managed integrated decep-

URI: http://hdl.handle.net/10125/50580
ISBN: 978-0-9981331-1-9

tion operations. It uses software product line (SPL) engineering, a well-known and widely-used technology in industry, to express deception strategies in a system [13, 14]. While deception can be applied in different layers of computation, we focused our approach on the application-level. It is at this level that a deception architecture can be best tuned to a specific software [15]. In particular, our contributions are: *(i)* a reusable solution based on SPL to produce multilevel deception defense, *(ii)* an architectural pattern to manage the phases of deception operations life cycle, *(iii)* a seamless integration of deception into system operations using aspect-orientation, and *(iv)* an architecture to manage adaptation on strategic and operational levels.

The remainder of this document is organized as follows. Section 2 presents the fundamental concepts of deception-based defense and SPL. Section 3 introduces our framework and describes the process to integrate deception mechanisms into a system based on SPL. Section 4 presents an instantiation of our framework using a general web application threat model. Finally, Section 5 describes related work, and Section 6 draws conclusions and suggests future work.

2. Background

2.1. Deception-based defense

The use of deception to enhance security in computer systems is a significant component of active defense. While traditional security mechanisms explicitly deny unauthorized accesses to system resources and obfuscate sensitive data, deception aims at leading attackers astray by drawing their attention to other pieces of decoy data and components specially crafted to mislead them. Typically, deception is not applied as a single mechanism for software systems defense. Instead, it is used orthogonally with techniques of denial and obfuscation, which brings particular opportunities for defense [2]. For example, adversaries are manipulated to spend more resources and time to accomplish tasks, which opens the possibility of learning about the attackers' modus operandi rather than merely monitor, detect, and block intrusions. Hence, deception techniques might enable the detection of zero-day attacks while reducing false-positives, a recurrent problem in traditional intrusion detection systems.

A key aspect of any deception operation relies on its ability to maintain a plausible story to adversaries. If a decoy resource is easily identified by adversaries, the operation will fail. Thus, the deception operation should be tightly integrated with the real system operation [16]. To incorporate deception operations into a system, both deception and system mechanisms must work in tandem to cope with the overall security of an application. The process of creating and operationalizing deception mechanisms can be complex, depending on how deception stories are related to each other, and how operational failures influence the operation [3]. For example, in a web application, decoy mechanisms can be associated with cookies, session control, CAPTCHA components, directories, passwords, etc. All of these mechanisms might be part of multiple stories. Also, solutions for problems that might occur during operation must be anticipated. Examples include solutions when the deception story is not received, or when the target discovers the deception, the story is not interpreted as intended, or the intended action is not taken.

Fig. 1 illustrates the general process of deception. It is divided into four phases: *Development, Deployment, Target Engagement, and Termination*.

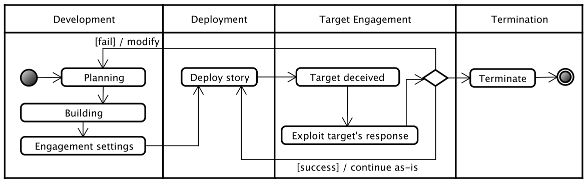


Figure 1. Deception process, adapted from [3]

Development. Planning is an interactive process that specifies the goals and objectives of the deception operation, the requirements and the target of the operation. The planner also identifies and analyzes risks associated with the deception operation, identifies responses for problems, and specifies the integration of the deception with other operations. During the Building process, a deception story is constructed, and feedback channels along with the termination plan are implemented. A deception story represents how the attacker will interact with the deception, which components will be placed during the operation and how these components interact with each other. Feedback channels provide the necessary information to analyze the operation and decide to continue or terminate. The termination plan specifies the activities performed during the finalization of the operation, such as clean up decoy components and bogus data from the system. Engagement settings determine how to exploit target actions, i.e., how to respond when a target engages the deception. Response to problems is delimited during the Engagement settings phase.

Deployment. The deception story is presented to the target in his observation field. At this point, the deception operation is out of the planner's control until some event (like a target interaction) suggests the operation is required to evolve.

Target Engagement. Engagement occurs when the target receives the deception story, accepts it (consider the story real), and, consequently, takes the intended action(s). Exploiting targets response represents the system functions performed when the target is deceived, which includes continuing the operation as-is, terminating the operation, or evolving it by returning to Planning.

Termination. Terminating includes the process of cleaning-up resources and tracks, and controlling the exposure of the operation.

2.2. Software product lines

SPL engineering refers to the methods, tools, and techniques for creating and maintaining a platform of common software systems from a shared set of software assets [17]. Its primary motivation relies on the fact that many parts of a software system share a common set of domain characteristics, meaning that SPLs stem from the need to facilitate mass customization of products. Instead of starting from scratch to produce a new product, we identify and describe throughout the development process where that products features may differ from others. A feature is a characteristic of a product that is visible to a stakeholder in some way [18].

A key aspect of SPL is managing the variabilities of the product line. Specifying variabilities can be done in several ways (from informal to formal approaches), but in general, feature models [18] and the Common Variability Language (CVL) [19] are popular choices. Feature models are tree-like structures that hierarchically decompose features using mandatory, optional or alternative refinements. CVL is a domain-independent language to specify and resolve variability in any (MOF)-based metamodel.

A typical SPL process comprises two phases: domain engineering and application engineering. Domain engineering models the domain concerns and establishes the foundation for deriving individual products, the remit of application engineering. In domain engineering, common features are intuitively identified and then variabilities are specified. The outcome of the domain engineering is a platform for a product line that aggregates necessary software artifacts (requirements, design, tests, etc.) that can be reused in other products. In SPL, each model is a product. The application engineering is the process responsible for generating product line applications from the platform established in the domain engineering. It exploits variabilities of the product line and ensures the correct binding of the variabilities according to the applications' specific needs.

3. SPL Framework for Adaptive Deception-based Defense

Our goal is to develop an appropriate reuse mechanism to develop multi-level deception defense, paying particular attention to the variability of deception tactics and their self-management. Fig. 2 presents the SPL process to integrate deception-based defense, which is compromised of three main phases: Domain Strategy Engineering, Application-level Strategy Engineering, and Weaving Process. To apply our process, developers only have to focus on the application-level strategy engineering. More specifically, a particular application threat model is selected and filtered with relevant threats chosen to be handled. Based on these threats, a configuration is derived automatically with the associated architecture. Finally, a weaving process containing composition rules adds deception capabilities to the system. Each element of our approach is detailed next.

The Deception Tactics Analysis provides the specification for a Deception Variability Model (DVM). Two base models provide common features for a DVM: the Deception Base Model providing base features concerning any deception tactic, and the Self-Adaptive Base Model providing base features concerning adaptation. The Deception Base Model comprises abstract and concrete features, as shown in Fig. 3.

Setup is an abstract feature encompassing all setup procedures to execute the deception tactic. *Configuration* includes *Action Policy* and *Alerts*. Action Policy refers to features describing the actions performed when a deception tactic is engaged, or when the deception fails for some reason. Similarly, Alerts refers to alarms that might be triggered when certain events occur, such as an interaction with some deceptive element or a relevant modification in the environment. *Services* will be refined into concrete functional features, i.e., those features that will realize the deception tactic. In our model, all deception tactics should be capable of being managed (*Management*) by activating (*Activation*) and deactivating (*Deactivation*) them. Optionally, a tactic can be temporarily enabled (*Enabling*) or disabled (*Disabling*), depending on the deception policy applied in the system. The last feature is *Termination*, which is composed of the features *Clean up* (for removing deception traces and control exposure of the tactic, so that it can be reused in further opportunities) and *Reorganization* (rearranging components to eliminate deceptive capabilities).

The Self-Adaptive Base Model (Fig. 4) represents a set of features that add self-adaptation capability to the system. Note that this is an abstract model that can be realized by different kinds of existing technologies

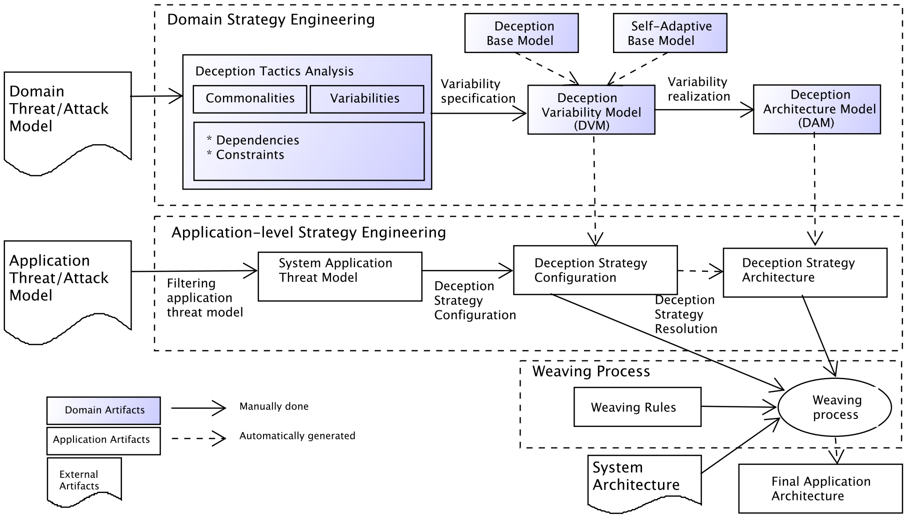

Domain Strategy Engineering
Deception Base Model
Self-Adaptive Base Model
Deception Tactics Analysis
Commonalities
Variabilities
* Dependencies
* Constraints
Domain Threat/Attack Model
Variability specification
Deception Variability Model (DVM)
Variability realization
Deception Architecture Model (DAM)
Application-level Strategy Engineering
Application Threat/Attack Model
Filtering application threat model
System Application Threat Model
Deception Strategy Configuration
Deception Strategy Configuration
Deception Strategy Architecture
Deception Strategy Resolution
Weaving Process
Weaving Rules
Weaving process
System Architecture
Final Application Architecture
Domain Artifacts
Application Artifacts
External Artifacts
Manually done
Automatically generated

Figure 2. SPL Process for Integrated Deception-based Defense

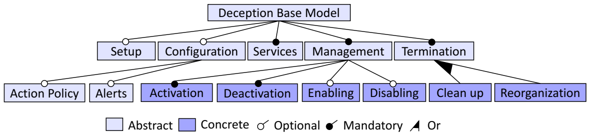


Figure 3. Deception Base Model

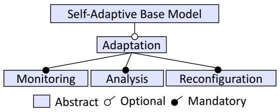


Figure 4. Self-Adaptive Base Model

(e.g. [20, 21]) or implemented specifically for an application. Adaptation is achieved by three basic features, inspired on MAPE-K loop [22]: *Monitoring*, *Analysis*, and *Reconfiguration*. Monitoring collects, normalizes, and correlates events occurring in the system or the environment (context). It can be accomplished through several techniques, using multiple channels to collect data. Examples include event log verification, code instrumentation, and external sensors. The analysis is where a decision to adapt is made. It consists of three steps: *(i)* aggregate and transform the systems state and context data into meaningful data according to adaptation decision policies, *(ii)* analyze synthesized data and rank eligible system configurations, based on variability and architectural models, *(iii)* decide on reconfiguring the system by updating architectural models with the corresponding generation of reconfiguration scripts. Finally, reconfiguration executes architectural modifications or parameter adjustment on the application to reflect a new system state. It is also responsible for the execution process, requiring some degree of access to recovery procedures in case of failures during adaptation.

The features of the Deception Variability Model are automatically inherited from the base models (Deception Base Model and Self-Adaptive Base Model). In the domain strategy engineering, a number of deception tactics can be available for a domain threat model. Let $DDT(d, dt)$ be a set of available domain deception tactics in a particular domain d with a declared threat model dt, and DVM a deception variability model, $DDT(d, dt) = (DVM_1, DVM_2, \ldots, DVM_n)$, we define a $DVM_x = (DBM, ABM, F_x)$, where DBM is a set of features related to the deception base model, ABM is a set of features related to self-adaptive base model, and F_x is a set of particular features of the deception tactic.

The next step is to map the variability model to the components of the deception architecture, i.e., the base model. This mapping is done by using bindings that link the variation points of the variability model to the UML design of the deception architecture model (DAM). Fig. 5 presents a partial view (high-level, conceptual) containing core components and dependencies of the deception architecture model. Deception Tactic is the component that initializes the tactic and communicates with other services and adaptation components of the system. Specific services provided by a tactic are extended from the D-Service component. Other features in the model are straightly associated to one single component in the architecture (e.g., Setup to D-Setup, Action policy to D-Action Policy, D-Alerts, D-Management, and so on). Adaptation is supported, as mentioned previously, by three core components: Monitor, Analysis, and Reconfiguration. We omit interfaces and other specialized components due to space constraints.

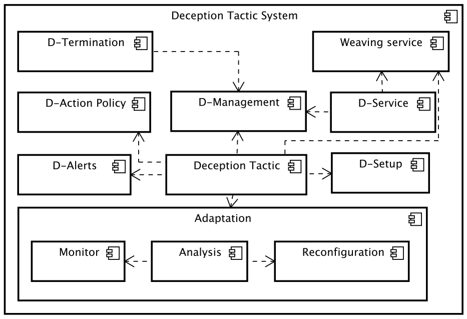


Figure 5. Deception architecture (excerpt)

3.1. Application-level Strategy Engineering

The Application-level Strategy Engineering (center Fig. 2) generates a valid configuration (or customization) of the variability model to add deception in a system. To increase flexibility in creating distinct configurations for a particular application, we consider a simple hierarchical structure composed by deception operation and strategies. An operation contains one or more strategies, and a strategy is composed of one or more tactics. Strategies own their specific composition rules, thus enabling a particular strategy (during the bootstrap or during adaptation) automatically activating all of its tactics. Fig. 6 illustrates a configuration based on strategies, where Deception Strategy 1 has two mandatory tactics (1 and 2), and Deception Strategy 2 has a mandatory (3) and an optional one (4).

A process to identify potential deception tactics is offered in [23]. Deception Strategy Configuration (center Fig. 2) selects a proper configuration, i.e., the tactics that will enhance the defense against the selected threats. Once these features are selected, a Deception Strategy Configuration is automatically created conforming to the DVM. Throughout the Deception Strategy Resolution, the Deception Strategy Architecture is automatically derived with components realizing the tactics.

To illustrate these steps, consider, for example, that a password cracking attack has been identified in a domain threat model (see detailed example in Section 4). During domain engineering, three deception tactics are included: Honeywords [6], Ersatzpasswords [7], and honeyaccounts (fake accounts). Honeywords and Ersatzpasswords are mutually exclusive tactics. This restriction must be expressed during domain engineering. During the application-level strategy engineering, a developer could choose to implement honeyaccounts and honeywords tactics and set specific features according to the variability points. For example, Enabling and Disabling features (Fig. 3) could not be part of the honeywords tactic but activated for honeyaccounts. Therefore, distinct tactics can be selected with different features to compose the final architecture.

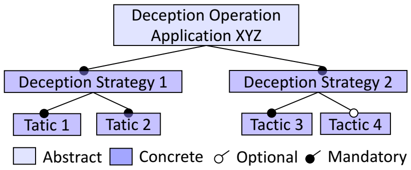


Figure 6. Deception Strategies

3.2. Weaving process

During the weaving process (bottom Fig. 2), the Final Application Architecture is constructed based on Weaving Rules that determine how to compose deception components into the System Architecture. Each feature in our model results in one or more components or interface/ports of components. Components can be expressed as black box entities, or as aspects [24], that will be composed to generate the final architecture. Weaving Rules determine which point of the system deception elements will actuate, considering common and specific features of a particular tactic. Such a weaving process allows keeping deception tactic concerns separated from system components, promoting reuse and facilitating work distribution among the developers teams. Also, as deception can be injected into the system dynamically, the system results more flexible.

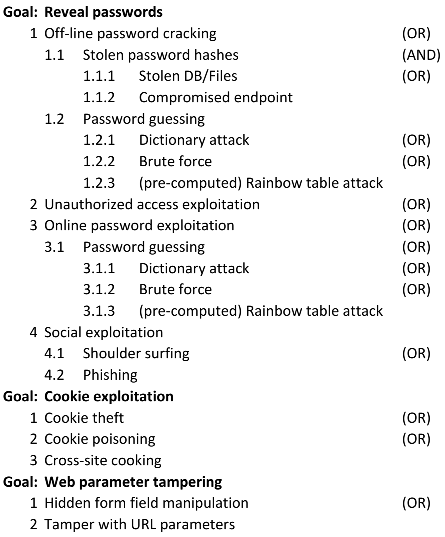
4. Illustrative Example

This section illustrates the applicability of our approach with a scenario from the web application domain.

4.1. Domain Strategy Analysis

The pervasive use of web applications offering critical services opens a significant range of security threats. We start by building (or selecting) a threat model for the web application domain. There are many methodologies and tools to create and represent a threat model, including agile methods and attack trees [25], misuse case graphs [26], CORAS [27], or STRIDE [28]. We use attack trees as they allow identifying scenarios uniquely, facilitating the matching process during application engineering. To identify attack scenarios, we use the OWASP attack catalog [29], a well-established source of web-application security information.

Fig. 7 presents an attack tree snippet containing three attackers goals: Reveal passwords, Cookie exploitation, and Web parameter tampering. To reveal passwords, attackers may perform Off-line password cracking (1), or Unauthorized access exploitation (2), or Online password exploitation (3), or Social exploitation (4). Cookie exploitation can be done by Cookie theft (1), Cookie poisoning (2), and Cross-Site cooking (3). Finally, Web parameter tampering could be achieved by Hidden form field manipulation (1), or Tamper with URL parameters (2). In our illustrative scenario, we will focus on three attacks: Off-line password cracking, Cookie Poisoning, and Hidden Form field manipulation.



Goal:	**Reveal passwords**	
1	Off-line password cracking	(OR)
1.1	Stolen password hashes	(AND)
1.1.1	Stolen DB/Files	(OR)
1.1.2	Compromised endpoint	
1.2	Password guessing	
1.2.1	Dictionary attack	(OR)
1.2.2	Brute force	(OR)
1.2.3	(pre-computed) Rainbow table attack	
2	Unauthorized access exploitation	(OR)
3	Online password exploitation	(OR)
3.1	Password guessing	(OR)
3.1.1	Dictionary attack	(OR)
3.1.2	Brute force	(OR)
3.1.3	(pre-computed) Rainbow table attack	
4	Social exploitation	
4.1	Shoulder surfing	(OR)
4.2	Phishing	
Goal:	**Cookie exploitation**	
1	Cookie theft	(OR)
2	Cookie poisoning	(OR)
3	Cross-site cooking	
Goal:	**Web parameter tampering**	
1	Hidden form field manipulation	(OR)
2	Tamper with URL parameters	

Figure 7. Attack Tree Snippet of Web Apps

During domain strategy analysis we can select or build a tactic using different levels of abstraction. As previously mentioned, we can use honeywords or ersatzpasswords as a deception-base defense tactic for off-line password cracking. Both tactics present different features that would be enacted in consonance with the application requirements during the application-level strategy analysis. Cookie poisoning is a classical attack where an attacker manipulates the value of a cookie to exploit some vulnerability in the application. Similarly, a hidden field can be shaped to catch some undesired behavior from the application.

A deception variability model for these attacks is shown in Fig. 8. The deception operation is composed of four tactics: Password tactics (Honeywords and Ersatzpasswords), Honeyaccounts, Honeyforms, and Honeycookies. Each tactic contains features derived from the Deception Base Model and the Self-Adaptive base Model. Honeywords illustrates the basic features (Setup, Configuration, Management, Services, Termination, and Adaptation). Basic features can be more or less restrictive conforming to the tactics rules. For example, Configuration is not a mandatory feature in the base model, but it is compulsory for this particular implementation of honeywords. Honeyforms considers injecting bogus hidden fields in a page. Similarly, Honeycookies inserts false cookies in the application, and Honeyaccounts creates fake user accounts in the system. Any activity of modifying a hidden field, a cookie value or use a honeyaccount (e.g., to login) indicates an abnormal behavior that increases the level of alert in the system.

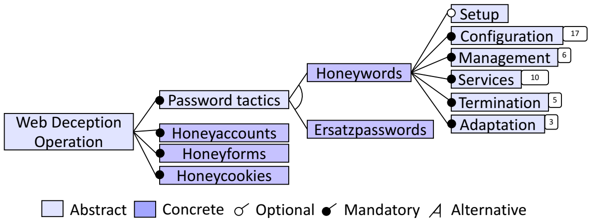


Figure 8. DVM for Web App domain

In our example, we will detail honeywords because they allow illustrating both basic and self-adaptive features. Honeywords is a deception-based mechanism proposed to detect the theft of a password repository. For each user account ui, a list Wi of k sweetwords is created where one password (the sugarword) is correct and $k - 1$ passwords (the honeywords) are fake. Sweetwords are generated by an algorithm $Gen(k)$, which output is the list $Wi = (wi, 1, wi, 2, \dots wi, k)$ and an index $c(i)$ representing the correct password. Alternatively, the procedure Gen allows an additional argument in the form of a user-supplied password pi. Thus, $Gen'(k, pi)$ guarantees that pi is a password within Wi, i.e. $pi = W_{i,C(i)}$. While a central server keeps all hashed sweetwords $(H(wi))$, the index of the correct password for each user is stored in an additional server, the honeychecker. Honeychecker provides a simple service interface that accepts $check(i, j)$ and $set(i, j)$ commands to verify whether $c(i) = j$ and set updates $c(i)$ have value j, respectively.

Adversaries who steal the password repository from the central server and reverse the hashed passwords should not be able to determine easily which one is the correct password. Any attempt to authenticate in the system with a honeyword triggers an event (e.g, an alarm to the administrators). The authentication scheme proceeds as follows: the input password $(H(p))$ is checked (as usual) against the hashed sweetwords. If the provided password does not match any sweetword, the login fails. If it matches a sweetword, the central server queries the honeychecker using a secure communication channel. If the checking finds the same index $(c(i) = j)$, the login process authenticates the user. Otherwise, a honeyword is being used and, depending on the system security policy various actions can occur: an alert can be triggered, blocking the account for verification (for example), and a deception message can be sent to the adversary (e.g., the system is currently under maintenance).

A critical process in the honeyword solution is the generation of the sweetwords (honeywords and sugarword). All sweetwords should have the same probability of being selected by an adversary (flatness). For example, the sugarword $L0b125$ with associated honeywords $L0b122, L0b124$ and $L0b123$, initially gives no clue to an attacker about which is the correct password. In general, honeywords generation methods are divided into two categories: preserved legacy user interface (UI) and modified user interface. Additional details about honeywords generation methods (e.g., Chaffing by tail tweaking, Chaffing by tail tweaking digits, Chaffing with a password model, and Take a tail) are presented in [6].

Honeywords provide four basic services (Honeyword building, Flatness checker, Check password, and Set password), as illustrated in Fig. 9. Honeyword building is responsible for generating the honeywords, while Flatness checker checks the level of flatness of the password. Check password and Set password are basically proxies for honeychecker functions. We assume the honeychecker is functional as an available external component. At least one algorithm to build the honeywords must be chosen (certain compositions are valid). Thus the relation OR (Typo-safe, Though nut, Chaffing by tweaking-digits, Chaffing by tail-tweaking, Password model, Take a tail) holds. Setup is an optional abstract feature; a concrete subfeature, for instance, would reset all passwords to be changed in the next user access, opening the opportunity to generate more honeywords. This task is performed only once, during the setup of the tactic.

Configuration considers Action Policy from the base model, and adds the K-coefficient feature, which represents the number of honeywords to be created for each user. We do not show the complete set of features in Fig. 9 (some are collapsed showing only the number of subfeatures), but we describe the key concepts of the model as follows. Management assumes enabling and disabling services as optional. Disabling is either refined into a XOR relation, considering disabling the tactic that monitors or removes monitoring capabilities. Action Policy is refined into features that specify what to do when the deception is engaged. This includes turn the computer off (turning down the entire system), shift the responses to a honeypot shifting (responses based on a honeypot system), and perform some action on the user account, such as password resetting and account blocking. Of course, these solutions have pros and cons that should be analyzed before activating them in a system. Adaptation, in our practical example, was left with basic features. However, subfeatures could be created to facilitate the work of activating them during applicationlevel strategy analysis. We do not show the mapping between features and components. Nevertheless, this is done in conformance with the deception architectural model presented in Fig. 5.

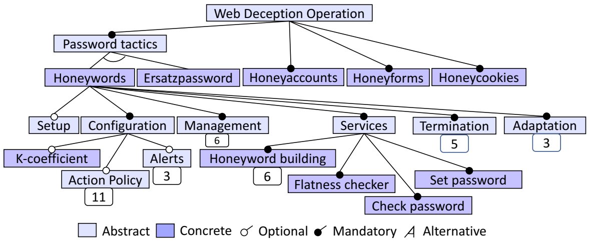


Figure 9. Services feature of Honeywords

4.2. Application-level Strategy Analysis

The application-level strategy analysis is intended to generate the final architecture of an application. Consider, for example, that a web-based financial system has been developed, and deception mechanisms are deemed to enhance the security of the application. The first step during this phase is to select the domain features that satisfy the application requirements. To generate a valid tactic configuration, the software architect/security engineering maps the application threat model containing selected threats to the features (tactics) of the DVM. The process compares specific threats with domain threats to filter which deception tactics could be selected.

After selecting the tactics that will be used in the application, the strategies can be composed. If only one strategy is necessary for the system, all selected tactics will be part of this strategy. Otherwise, different compositions with their respective constraints can be considered. Fig. 10 illustrates a deception strategy configuration for our illustrative financial system. Strategy 1 is compounded by honeywords and honeycookies, and Strategy 2 contains honeywords, honeycookies, honeyforms, and honeyaccounts. Honeywords are mandatory in both strategies, but honeycookies are optional only in strategy 1.

Once the deception strategy is set, the model can be refined with specific features for the application. In our example, lets use Strategy 1 to exemplify the feature refinement. The feature Alerts (from Configuration) can be refined into Admin SMS, Admin email, and Event log, representing alerts by SMS to administrators, alerts by mail and event log entry, respectively. Also, adaptive capabilities are required in honeywords since the password policy can change, what will demand the reconfiguration of algorithms used to generate the honeywords. For example, if the policy changes to not allow consecutive digits or personal data (e.g., birth date) into passwords, the tactic adapts to switch a better algorithm to keep flatness. In other circumstances, the system can adapt to change k-coefficient in response to an emergent need of increasing the entropy of the honeywords (adding honeywords) or save space in the disk (reducing honeywords). Monitoring is performed directly by aspects that identify whether a password policy has changed. Thus, a particular monitoring feature could be created to represent this capability.

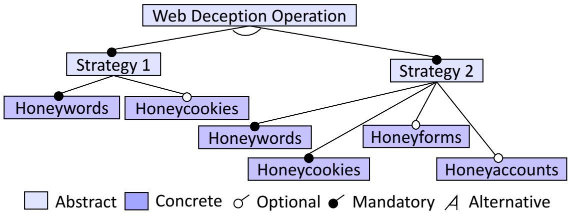


Figure 10. Deception Strategy Configuration

After configuring the deception strategies, Deception Strategy Architecture is derived in conformance with the Deception Architecture Model. For those features that are not part of the DVM, i.e., the refined features, we apply a one-to-one transformation, where the feature is translated into an aspect. Based on the composition rules, the Deception Strategy Configuration, Deception Strategy Architecture, and the System Architecture are composed, forming the Final Application Architecture. Fig. 11 illustrates the generated architecture. Due to space constraints, we omit the adaptation components, ports and some interfaces from the model. Also, we focus on Service, Management, and Adaptation features to illustrate how the architecture is generated. The Honeyword tactic is the central component of the architecture. It is modeled as an aspect responsible for coordinating the services and the management of the tactic. Also, this aspect crosscuts two system components: Account Manager and Authentication.

Account manager is responsible for handling user data, including passwords. Authentication is in charge of the verification of credentials. D-Management handles two types of services: basic and specific. Enabling (IEnableTactic) and disabling operations (IDisableTactic) are considered basic services as they can be used regardless of the tactic being implemented. Specific services are particular for the solution being implemented. In the honeywords example, services like checking and setting password indexes (ICheckPassword and ISetPassword), checking password flatness (IFlatnessChecker), and honeyword generation are specific services. In our illustrative example, we instantiate Chaffing by tail-tweaking and Type-Safe, and also instantiate SMS Alert and Event Log components.

The Honeyword Adaptation component contains high-level adaptation components, where Security Policy Monitor is an aspect that crosscuts the system component Security Policy to identify changes in password policies. Tactic Analysis is responsible for analyzing changes in security policies, identifying a plan from the available ones into the plan description and selecting an appropriate one to be executed with the assistance of the Plan Selector component. A plan describes possible variations (structural or parametric) that can be applied in the system.

5. Related work

Several approaches deal with the integration of deception into security operations. A process to incorporate deception into security operations is proposed in [30], and a goal-driven approach to specifying deception tactics is proposed in [23]. The former presents a high-level process to integrate deception operations, and the later specifies strategies and tactics as features. Neither of these approaches detail how to resolve the feature model at architectural level. Our proposal addresses this issue.

In [31], an aspect-oriented approach is used to deploy honeytokens within a database management system. The architectural design considers operations and pointcuts in a single aspect that intercepts database queries to generate, manage, detect and distribute honeytokens. This work is focused on automating the creation of honeytokens in databases. Our work incorporates deception mechanisms as functional components of the application. In [10] is proposed an event-based mechanism to detect abnormal behavior and a generic architecture based on wrappers to implement deception. In contrast, our approach proposes a SPL perspective to the problem of integrating deception into

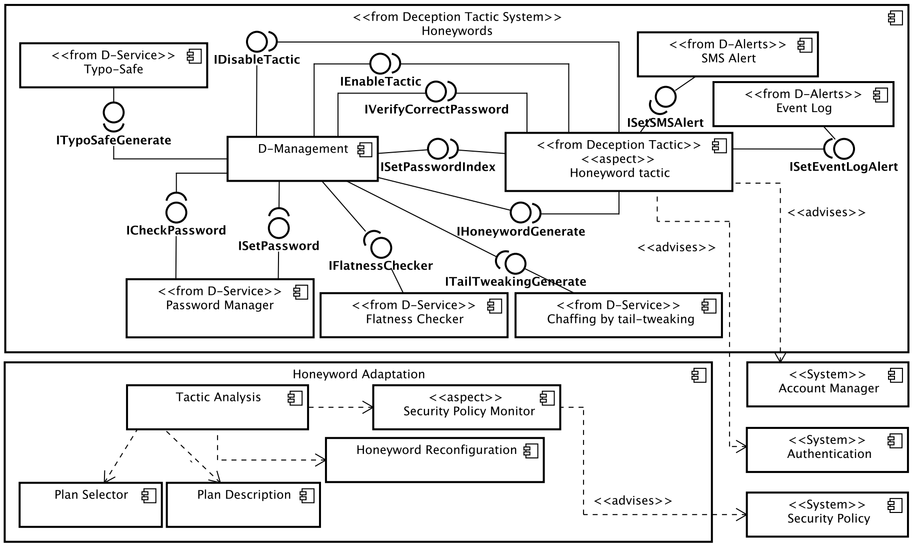


Figure 11. Final Application Architecture (excerpt)

applications.

Other SPL approaches [32, 33] are employed for traditional security mechanisms and frameworks are proposed to build a security core asset repository based on requirements security and Common Criteria (ISO/IEC 15408) and the ISO/IEC 17799. Similarly, [34] presents a design theory and artifacts to represent and configure SPLs. This work is focused on varying security and functionality from the requirements to the SPL architecture. In [35], SPL and aspect-oriented programming are suggested to close the gap between the specification of a security policy and its deployment. These works are based on SPL and involve security, but they are not focused on deception-based solutions.

6. Conclusions and future work

We presented an SPL framework to create adaptive deception-based defense integrated to applications. The result is a repository of security core assets based on deception containing the main features to manage the operation life cycle, including initialization, termination, and adaptive capabilities. Following our approach, deception capabilities are defined in terms of basic and specific features that are mapped to a base architecture model. Engineers focus on selecting a set of threats or attack vectors to create a specific configuration representing a set of deception strategies for a system. Strategies and tactics constitute a simple structure to group mechanisms designed to trap adversaries. Basic features provide support for the management of each tactic, involving setup, monitoring, response policies, initialization and termination procedures. We applied our framework in the integration of Honeywords to illustrate how a non-trivial deception-based solution can be incorporate into existing security mechanisms.

While our approach intends to promote reuse and seamless integration of deception in a system, it does not ensure that the system itself will be more secure. Each tactic has specific goals and, as any other security mechanism, may incur in additional vulnerabilities that should be identified and managed as soon as possible in the software development life cycle. It must also be taken into consideration that the use of deception components may overload the system operation, requiring careful analysis before deploying any of these solutions. As part of our future work, the framework will be evaluated using more domains and additional threat models to reason about incurring trade-offs of incorporating deception into a system. Also, we intend to propose the framework to facilitate the implementation of honeypots and honeynets, which is a recurrent issue in the literature.

Acknowledgments

We are grateful to CAPES foundation and NOVA LINCS Research Laboratory (Ref. UID/CEC/04516/2013) for their support in the development of this work.

References

[1] D. S. Livingstone, “Why We Lie. The Evolutionary Roots of Deception and the Unconscious Mind,” *The Journal of Analytical Psychology*, vol. 50, no. 5, pp. 715–716, 2005.

[2] K. E. Heckman, F. J. Stech, R. K. Thomas, B. Schmoker, and A. W. Tsow, *Cyber Denial, Deception and Counter Deception*. Springer, 2015.

[3] J. J. Yuill, *Defensive Computer-security Deception Operations: Processes, Principles and Techniques*. PhD thesis, 2006.

[4] L. Spitzner, *Honeypots: Tracking Hackers*, vol. 1. Addison-Wesley Reading, 2002.

[5] W. Wang, J. Bickford, I. Murynets, R. Subbaraman, A. G. Forte, and G. Singaraju, "Catching the wily hacker: A multilayer deception system," in *Sarnoff Symposium (SARNOFF), 2012 35th IEEE*, pp. 1–6, IEEE, 2012.

[6] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 145–160, ACM, 2013.

[7] M. H. Almeshekah, C. N. Gutierrez, M. J. Atallah, and E. H. Spafford, "ErsatzPasswords: Ending Password Cracking and Detecting Password Leakage," in *Proceedings of the 31st Annual Computer Security Applications Conference*, pp. 311–320, ACM, 2015.

[8] S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Booby trapping software," *Proceedings of the 2013 workshop on New security paradigms workshop - NSPW '13*, pp. 95–106, 2013.

[9] F. De Gaspari, S. Jajodia, L. V. Mancini, and A. Panico, "AHEAD: A New Architecture for Active Defense," *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense - SafeConfig'16*, pp. 11–16, 2016.

[10] B. Michael, M. Auguston, N. Rowe, and R. Riehle, "Software Decoys: Intrusion Detection and Countermeasures," *Proceedings of the IEEE Workshop on Information Assurance*, vol. 0001, no. June, pp. 130–138, 2002.

[11] M. L. Bringer, C. A. Chelmecki, and H. Fujinoki, "A Survey: Recent Advances and Future Trends in Honeypot Research," *International Journal of Computer Network and Information Security*, vol. 4, no. 10, pp. 65–77, 2012.

[12] F. Cohen, "A Framework for Deception," *National Security Issues in Science, Law, and Technology*, p. 123, 2007.

[13] P. C. Clements, *Software architecture in practice*. 2002.

[14] F. J. der Linden, K. Schmid, and E. Rommes, *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.

[15] N. C. Rowe and J. Rrushi, *Introduction to Cyberdeception*. Springer, 2016.

[16] M. H. Almeshekah, *Using deception to enhance security: A Taxonomy, Model, and Novel Uses*. PhD thesis, Purdue University, 2015.

[17] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[18] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," tech. rep., DTIC Document, 1990.

[19] . Haugen, A. Wąsowski, and K. Czarnecki, "CVL Common Variability Language," in *Proceedings of the 17th International Software Product Line Conference on - SPLC '13*, p. 277, 2013.

[20] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture- Based Self-Adaptation with Reusable Infrastructure," *Computer*, pp. 46–54, 2004.

[21] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Autonomic computing through reuse of variability models at runtime: The case of smart homes," *Computer*, vol. 42, no. 10, pp. 37–43, 2009.

[22] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[23] C. D. Faveri, A. Moreira, and V. Amaral, "Goal-Driven Deception Tactics Design," in *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pp. 264–275, 2016.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," *ECOOP'97 Object-Oriented Programming*, vol. 1241/1997, pp. 220–242, 1997.

[25] S. Mauw and M. Oostdijk, "Foundations of attack trees," in *International Conference on Information Security and Cryptology*, pp. 186–198, Springer, 2005.

[26] G. Sindre and A. L. Opdahl, "Eliciting security requirements with misuse cases," *Requirements engineering*, vol. 10, no. 1, pp. 34–44, 2005.

[27] F. den Braber, T. Dimitrakos, B. A. Gran, M. S. Lund, K. Stølen, and J. . Aagedal, "The CORAS methodology: model-based risk assessment using UML and UP," *UML and the Unified Process*, pp. 332–357, 2003.

[28] Microsoft, "The STRIDE Threat Model. URL: https://goo.gl/G5Fzbi," 2002.

[29] "Open Web Application Security Project (OWASP). URL: https://www.owasp.org," 2012.

[30] M. H. Almeshekah and E. H. Spafford, "Planning and Integrating Deception into Computer Security Defenses," in *Proceedings of the 2014 workshop on New Security Paradigms Workshop - NSPW '14*, pp. 127–138, ACM, 2014.

[31] K. Padayachee, "Aspectising honeytokens to contain the insider threat," *IET Information Security*, vol. 9, no. 4, pp. 240 – 247, 2015.

[32] D. Mellado, E. Fernández-Medina, and M. Piattini, "Towards security requirements management for software product lines: A security domain requirements engineering process," *Computer Standards & Interfaces*, vol. 30, no. 6, pp. 361–371, 2008.

[33] D. Mellado, E. Fernández-Medina, and M. Piattini, "Security requirements engineering framework for software product lines," *Information and Software Technology*, vol. 52, no. 10, pp. 1094–1117, 2010.

[34] V. Myllärniemi, M. Raatikainen, and T. Männistö, "Representing and Configuring Security Variability in Software Product Lines," in *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, pp. 1–10, 2015.

[35] J.-M. Horcas, M. Pinto, and L. Fuentes, "Closing the gap between the specification and enforcement of security policies," in *International Conference on Trust, Privacy and Security in Digital Business*, pp. 106–118, Springer, 2014.