In 2012 International Conference on Web Information Systems Engineering (WISE 2012), Nov 28-30 2012, Paphos, Cyprus, LNCS, Springer.

VAM-aaS: Online Cloud Services Security Vulnerability Analysis and Mitigation-as-a-Service

Mohamed Almorsy, John Grundy, and Amani S. Ibrahim

Computer science and software engineering center
Swinburne University of Technology
Melbourne, Australia
[malmorsy, jgrundy, aibrahim]@swin.edu.au

Abstract. Cloud computing introduces a new paradigm shift in service delivery models. However, the potential benefits reaped from the adoption of this model are threatened by public accessibility of the cloud-hosted services and sharing of resources. This increases the possibility of malicious service attacks. Existing cloud platforms do not provide a means to validate the security of offered cloud services. Moreover, the public accessibility of cloud services increases the potential for exploitation of newly discovered vulnerabilities that usually take a long time to discover and to mitigate. We introduce VAM-aaS, Vulnerability Analysis and Mitigation as-a-service, as a novel, integrated, and online cloud-based security vulnerability analysis and mitigation service. VAM-aaS performs online service analysis to pinpoint new vulnerabilities and weaknesses. It then uses this information to generate security control configuration scripts to block these discovered security holes at runtime. Our approach is based on a new vulnerability signature and mitigation-actions specification approach. We introduce our approach, describe key implementation details, and describe an evaluation of our prototype on a set of .NET benchmark applications.

Keywords: SaaS Security; Vulnerability Analysis; Vulnerability Mitigation

1 Introduction

The cloud computing model [1] introduces a new paradigm shift in computing platforms with an emphasis on increasing business benefits. The cloud model is based on service outsourcing of application hosting on third-party platforms outside of the enterprise network perimeter. It uses a new pay-as-you-go payment model where customers can rent services occasionally and pay only for amount of resources they use. However, the cloud model also introduces new opportunities for attackers to exploit, such as publicly accessible valuable business services. Adopting the multi-tenancy model increases the exploitability of service vulnerabilities because one of the service tenants, who have privileged access to the cloud service, may be a malicious user. This means that they can exploit complicated vulnerabilities that require higher privileges rather than being a public user. Moreover, the number of newly discovered vulnerabilities is increasing rapidly. Web applications, the most prominent application

© Springer-Verlag Berlin Heidelberg 2011

delivery model used in SaaS applications, continue to make up the largest percentage (75%) of the total reported vulnerabilities over the last three years [24].

Commercial vulnerability scanners such as IBM-AppScan, HP-Web inspect, McAfee tools focus mainly on black-box vulnerability analysis to avoid being limited to specific programming languages or platforms. However, none of these scanners cover all known vulnerability types [2]. On the other hand, existing research efforts [3-8] focus on discovering specific vulnerability types including SQLI [9, 10], XSS [9, 11, 12], or input sanitization [13, 14] using static analysis [3, 15], dynamic analysis [9], or hybrid techniques [16, 17].

The key problems with these efforts include: they provide specific techniques for specific vulnerability types; the techniques apply only to specific platforms or languages; and they do not usually support analysis for new vulnerability types. On the other hand, these limitations are key requirements for cloud services vulnerability analysis tools. An ***online*** vulnerability analysis approach that supports locating well-known as well as new vulnerabilities without waiting for new tool patches is a must to have in the cloud computing environment.

Mitigating application vulnerabilities is usually done manually by modifying application source code and deploying new patches; however, this takes a long time as shown in Fig. 1. This lagging time between vulnerability detection and patch means that the service remains vulnerable to security breaches exploiting such vulnerabilities. The possibility of vulnerability exploitation increases dramatically in the cloud, given the public accessibility of the cloud services and the sharing of services with multiple tenants. Thus, the cloud computing model requires an online vulnerability patching approach that can block such vulnerabilities once reported.

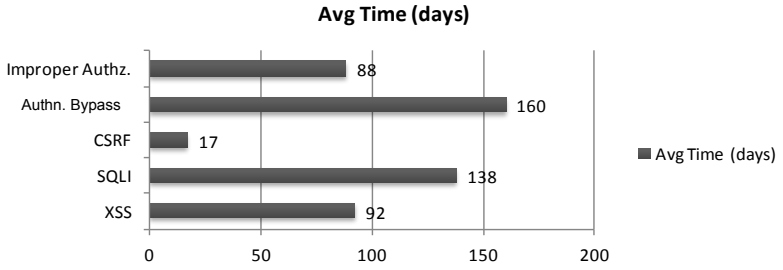

Fig. 1. Average time to fix security vulnerabilities (in days)

In this paper, we introduce VAM-aaS, a new integrated solution to cloud-based services vulnerability analysis and mitigation problems. Our approach is based on a new formal approach to specifying vulnerability signatures as well as enumerated mitigation actions to block such vulnerability. Vulnerability signatures are specified as invariants. When one is matched in a target application it means that the specified vulnerabilities exist in the application or service under analysis. We adopt Object Constraint Language (OCL), as a declarative and formal language based on first order logic and set theory, to capture vulnerability signatures. Such signatures are validated against a comprehensive system description meta-model (represents language semantics) covering most of the object oriented program concepts and entities. We developed a vulnerability analysis service that locates OCL-based vulnerabilities' signa-

tures in a given SaaS application source code. To support the location of new types of vulnerabilities, security experts need to update the vulnerability analysis service repository with OCL-signatures for the new vulnerability. Our vulnerability analysis component can be used to analyze both cloud services and traditional web applications.

The vulnerability mitigation actions specify a set of security solutions that can be used to provide "virtual patching" of the discovered vulnerabilities reported by the analysis component. They also specify configurations and rules that should be applied when activating security controls. We have developed a vulnerability mitigation component that uses these vulnerability mitigation actions to plug-in specified mitigation security controls within the target vulnerable service or application online. Our mitigation component is not limited to or hardcoded to specific security controls. It depends on a simplified security interface that security controls should satisfy in order to be integrated with our mitigation component. Thus, new security controls can be plugged into the system vulnerable entity at runtime to mitigate the vulnerability.

We evaluated our approach and its prototype realization service in capturing the well-known Top10 vulnerabilities reported by OWSAP [25]. We have validated our toolset in locating and mitigating these vulnerabilities on a set of open source web applications.

In Section 2, we describe our approach, vulnerability signatures, and mitigation actions. In Section 3, we describe our OCL-based vulnerability analysis component. In Section 4, we describe our vulnerability mitigation component. In Section 5, we describe our prototype implementation details. In Section 6, we discuss our experimental evaluation and results. In Section 7, we discuss the implications of our work and key directions for further research. In Section 8, we review the key related work.

2 Our Approach

Our security vulnerability analysis and mitigation approach is based on (i) a formalized vulnerability signature and potential mitigation actions specification; (ii) a vulnerability analysis tool that performs OCL-based vulnerability signature-based program analysis; and (iii) a vulnerability mitigation component that blocks service or application security vulnerabilities by generating configuration and integration scripts that integrate security controls at the application or service reported vulnerable points. In Fig. 2, we summarize the possible interactions between the vulnerabilities definition repository, analysis and mitigation components, applications or services, and the hosting service (Web Server, Operating System, etc).

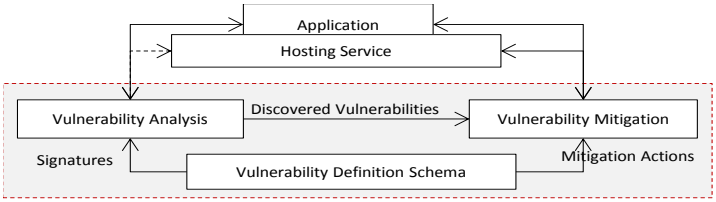

Fig. 2. VAM-aaS Key components, relations and possible interactions

2.1 Vulnerability Signature Specification

Existing software security weakness, or vulnerability definitions, in the Common Weakness Enumeration (CWE) [27] database help in understanding the nature of a given vulnerability. However, these vulnerabilities' definitions are informal. This requires manual analysis (by security experts) to locate such vulnerabilities in the applications under analysis.

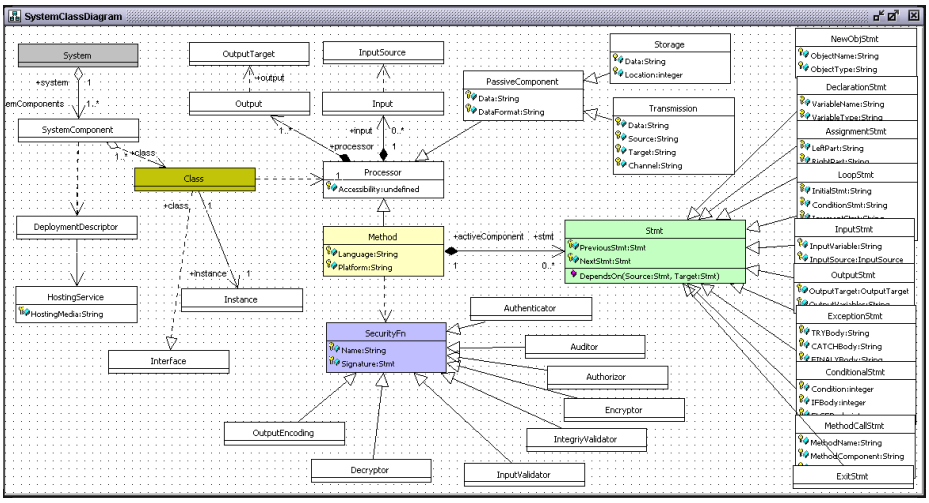

Fig. 3. System description meta-model used to specify OCL-based vulnerability signatures

Formalizing these descriptions – i.e. the vulnerability signatures – allows automation of the vulnerability analysis process. Ideally, a formal vulnerability signature should be specified at an abstract level far from the source code and programming language details, enabling locating of possible vulnerability instances in different programs written in different programming languages. We use the Object Constraint Language (OCL) as a well-known, extensible, and formal language to specify semantic rather than syntactical signatures of security weaknesses. To support specifying and validating OCL-based vulnerabilities' signatures, we have developed a system-description meta-model, shown in Fig. 3. This model is inspired from our analysis of the nature of the existing security vulnerabilities. It captures the main entities in any object-oriented program and relationships between them including components, classes, instances, inputs, input sources, output, output targets, methods, method bodies, statements e.g. if-else statements, loops, new objects, etc. Each entity has a set of attributes such as method name, accessibility, variable name, variable type, method call name. This model helps conducting semantic analysis of the specified vulnerability signatures.

Some vulnerabilities require checking the existence of a security control that authenticates, authorizes, audits, etc. at specific locations in the program e.g. before a critical method call, their use should be authenticated and authorized. Fig. 3 shows security functions as part of the system model. They inherit from the Method entity and thus can be checked in method call statement – i.e. check if the invoked method is

a security method or not. An analysis tool should have different profiles for different languages and platforms (ASP.Net, PHP, C#, Java, etc.). Thus vulnerabilities with signatures containing input source or output target can be interpreted differently based on the program platform or language. Moreover, security authentication, authorization, sanitization and other functions will be interpreted according to the target system and the underlying platform. In case of custom security functions, system developers have to manually mark their security functions. Table 1 shows examples of vulnerability signatures specified in OCL and using our system description model (Fig. 3).

Table 1. Examples of OCL-specified vulnerability signatures

Vul.	Vulnerability Signature
SQLI	Context Method Inv SQLICheck: self.Statements->exists(S | S.StatementType = 'MethodInvocation' and S.MethodName = 'ExecuteSQL' and S.Parameters.exists(P | self.IsTainted(P.ParameterName) = true)
XSS	Context Method Inv SQLICheck: self.Exists(S | S.StatementType = 'Assignment' and S.RightPart.Contains(InputSource) and S.LeftPart.Contains(OutputTarget))
Authn. Bypass	Context Method Inv SQLICheck: self.IsPublic == true and self->Exists(S | S.StatementType = 'MethodInvocation' and S.IsAuthenitcationFn == true and S.Parent == IFElseStmt and S.Parent.Condition.Contains(InputSource))
Improper Authz.	Context Method Inv SQLICheck: self.IsPublic == true and self.Contains(S| S.Exists(X| X.StatementType = 'InputSource' and X.IsSanitized = false or X.IsAuthorized == False)

SQLI: any method statement *"S'* of type "*MethodInvocation*" where the callee function is *"ExecuteQuery"* and one of the *parameters* passed to it, is assigned to *"identifier"* coming from one of the input sources. Taint analysis "*IsTainted*" can be defined as an OCL function that adds every variable assigned to a user input parameter to a suspected list.

XSS Signature: any method statement *"S"* of type assignment statement where left part is of type "*output target*" e.g. text, label, grid, etc. and right part uses input from the input sources or tainted identifier as just discussed.

Authn. Bypass: any public method that has statement "*S*" of type "*MethodInvocation*" where the callee method is marked as Authentication function while this method call can be skipped using user input as part of the bypassing condition.

Improper Authz.: any public method that has statement "S" that uses input data X without being sanitized, authorized.

These exemplar signatures focus on static vulnerabilities signatures and do not consider security solutions applied beyond the system source code either using proxies to filter SQL queries or using security controls deployed on the web server as an http handler. These can be handled by appending a dynamic signature forming a sequence of OCL constraints to be checked – i.e. to check if requests and/or responses contain certain vulnerability pattern. Weak signatures result in more false positives, which may annoy developers, or more false negatives, which may harm customers.

2.2 Mitigation Actions

Discovered application or service security vulnerabilities can be mitigated by different approaches including modifying application source code to block the identified

problems (patches). However, this solution is hard to use in the cloud model as it may take a long time to deliver patched versions, as shown in Fig. 1. One solution is to use Web application firewall (WAF) to filter requests and responses that exploit such vulnerabilities. However, WAF has many limitations. These include does not helping with output validation, cryptographic storage, and mitigating improper authorization.

Table 2. Examples of vulnerability mitigation actions

Vul.	Security Control	Entity Level
SQLI	Input sanitization	Method level
XSS	Input encoding	Component level
Authn. Bypass	WAF	Component level
Improper Authz.	Authorization	Method Level

We introduce a new approach that supports integration of different security controls including identity management, authentication controls, authorization controls, input validation, output encoding, WAF, cryptography controls, etc. In our approach, each vulnerability mitigation action specifies a security control type or family to be used in mitigating the related vulnerability, its required configurations, and application or service entity where the security control will be integrated (e.g. hosting service – webserver or operating system, components, classes, and methods). Thus, a reported SQLI vulnerability in a method (M) that belongs to component (C) can be mitigated by adding input sanitization control (Z) on component (C) that removes SQL keyword from every single request to the method (M). In Table 2, we show examples of mitigation actions for some of the known security vulnerabilities. These actions should be specified in XML and included as a part of the formalized vulnerability definition.

3 Our OCL-based Vulnerability Analyzer

Given that vulnerability signatures are now formally specified using OCL, the static vulnerability analysis component simply traverses the given program looking for code snippets with matches to the given vulnerabilities' signatures. The architecture of our formal and scalable static vulnerability analysis component, as shown in Fig. 4, is based on our formalized vulnerability signature concept.

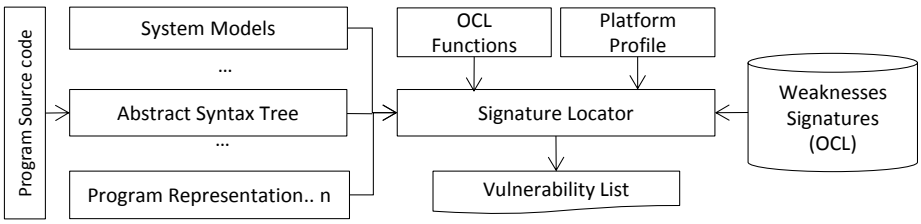

Fig. 4. OCL-based vulnerability analysis component

Program Source Code: our analysis approach work on source code level. In case system binaries only available (dlls, or exes), we use de-compilation techniques to reverse engineer source code from the application to be analyzed.

Abstract Program Representation: Source code is transformed into an abstract syntax tree (AST) representation. This abstracts language-specific source code details

away from specific language constructs. Extracting source code AST requires using different language parsers (currently support C++, VB.Net and C#). Then, we perform more abstract transforming from AST to system description model that conforms to the model introduced in Fig. 3.

OCL Functions: represent a library of predefined functions that can be used in specifying vulnerability signatures and in identifying matches to these signatures. This includes control flow, data flow, string patterns, program taint analysis, etc.

Signature Locator: This is the main component in our vulnerability analysis tool. It receives the abstract service or application model and outputs the list of discovered vulnerabilities in the given system along with their locations in code. At analysis time, it loads the platform (C#, VB, PHP) profile based on the details of the program under analysis. Then, it loads the existing weaknesses defined in the weaknesses' signatures database, based on the target implementation program platform or language. The signature locator transforms these signatures into C# methods that check different program entities based on the specified vulnerability signature.

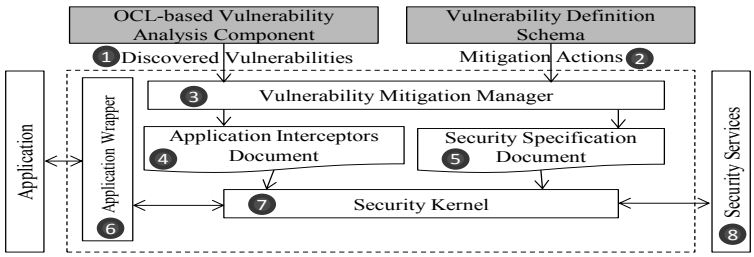

Fig. 5. Vulnerability Mitigation Component

4 Vulnerability Mitigation Component

The analysis component outputs a list of the newly discovered vulnerabilities in each of the cloud hosted SaaS applications (Fig. 5-1). Each entry in this list has a service or application "vulnerable entity" (e.g. a method, class, or component), along with a list of discovered vulnerabilities for this entity. Given this list of vulnerabilities, the security vulnerability mitigation manager queries the vulnerability definition schema database (Fig. 5-2) to retrieve the appropriate actions to be taken in order to mitigate each of such reported vulnerabilities. Examples of such retrieved actions are shown in Table 2. Using these two lists (vulnerable entities and mitigation actions), the vulnerability mitigation manager (Fig. 5-3) decides the patching level (component level, class level, or method level) using e.g. HttpModules, object interceptor using dependency injection, or method level interception using dynamic weaving AOP respectively. These details are maintained in a security point-cut specification document for each application (Fig. 5-4). Moreover, the mitigation manager uses the registered security services' properties to decide which security service realizes what security control type specified in the mitigation action – e.g. the identity and access management control is currently (on this platform) realized by CA identity manager. Finally, the vulnerability mitigation manager updates the security specification document (Fig. 5-5) with the list of actual security services to be triggered whenever the application

receives request to every vulnerable resource. The application wrapper (Fig. 5-6) is responsible for intercepting requests to entities specified in the application interceptors' document – i.e. vulnerable entities. These requests will be redirected to the security kernel (Fig. 5-7). The security kernel queries the security specification document to get the security controls or services to be enforced to secure the requested resource. Then, it generates a set of calls to these security services (Fig. 5-8). When these services return, the security kernel returns the control back to the called resource.

To automate the integration of security services with our mitigation component, which means being implicitly integrated with the managed service or application, we have developed a simple common security interface. Security controls' connectors implement this interface in order to support integration with our mitigation service. For example, an authentication security control should implement AuthenticateUser and IsAuthenticated functions; an input validation control should implement the ValidateInput function; and an output encoder should implement the EncodeOutput function. Security controls used in the vulnerability mitigation process can be part of a standard security controls library provided by the service provider or the cloud provider. Moreover, they may be external security controls hosted on other cloud platforms. In our prototype, we use the OWASP security controls library.

5 Implementation

We developed a GUI to assist security experts in capturing vulnerability signatures' in OCL. This provides vulnerability signature editing, validity checking, and testing these signatures' specifications on simple target applications. We use an existing OCL parser to parse and validate signatures against our system description meta-model. Once validated, the vulnerability signature is stored in our weakness signatures database. To parse the given program source code and generate a system abstract model, we use ***NReFactory*** .NET parser Library [27], which parses source code and generates its corresponding AST (it supports VB.Net and C#. We are currently working on parsers for PhP and Java). Applications without source code - i.e. only binaries are available – are decompiled using ***ILSPY***. This is currently supported for C# and VB.NET. We developed a class library to transform the generated AST into a more abstract (summarized) representation that conforms to our system description model. Our signature locator has an OCL translator that translates a given OCL signature into a corresponding C# class with a signature matching method that checks the passed in system entity looking for matches to specified signatures.

The OCL functions library maintains a set of functions that extend the system description meta-model entities capabilities and can be used during the vulnerability analysis phase. This includes control-flow analysis (CFA), data-flow analysis (DFA), and tainted-data analysis. These functions can be extended with further analysis functions based on future vulnerability analysis needs. The OCL to C# transformer performs a transformation for these functions as well as new OCL signatures once defined. Program slicing and taint analysis techniques (core techniques in program and security analysis area) can be easily captured in OCL. Platforms' profiles are specified in XML documents that contain information about specific platforms' details. It is used to set the context of the signature locator according to the system platform.

The vulnerability mitigation component was developed using C#. It uses the Microsoft Unity application block to support configurable runtime dependency injection. This enables injecting interceptors on class and method level. To support legacy services, we use the Yiihaw static AOP tool [28]. This enables injecting aspects into legacy code. Such aspects and interceptors redirect requests and responses to a default security handler class library (we call a Security Enforcement Point, or SEP). The SEP is responsible for injecting calls to security controls at runtime based on the specified mitigation actions.

6 Experimental Evaluation

The key objectives of these experiments was to assess the soundness of our VAM-aaS in capturing different vulnerabilities' signatures, detecting these vulnerabilities in given applications, and in mitigating them. We apply the OCL-based vulnerability signatures examples and mitigation actions discussed in Section 2.We selected seven web-based, open source applications developed using ASP.NET and MVC as a benchmark to evaluate our approach. These applications cover a wide business spectrum including: Galactic, an ERP system; SplendidCRM, an open source CRM; KOOBOO, an open source Enterprise CMS; BlogEngine, an open source ASP.NET 4.0 blogging engine; BugTracer, an open-source, web-based bug tracking application; NopCommerce, an open-source eCommerce solution; and Webgoat, developed by OWSAP for security testing purposes. In Table 3, we summarize these benchmark applications' characteristics: number of downloads, LOCs, files, classes, methods, and time (ms) to extract system model from source code (using source code AST).

To assess the effectiveness of our approach we use precision, recall and f-measure to measure our approach's soundness and completeness. These metrics depend on basic measures shown in Table 4. The analysis component results are compared with the actual vulnerabilities discovered by manual analysis using existing open source vulnerability analysis tools. As shown in Table 4-A, True Positive (TP) counts number of vulnerabilities correctly discovered by the analysis component, False Positive (FP) counts number of vulnerabilities incorrectly reported as vulnerability, False Negative (FN) counts number of vulnerabilities missed by the analysis component. As for the mitigation component (Table 5-B), the TP counts number of vulnerabilities that have been correctly blocked, FN counts number of vulnerabilities failed to block, and FP counts entities that have been secured without being a vulnerability.

Table 3. Benchmark applications properties

Benchmark	Downloads	KLOC	Files	Classes	Method	Model time
Galactic	-	16.2	99	101	473	187
SplendidCRM	>400	245	816	6177	6107	765
KOOBOO	>2,000	112	1178	7851	5083	78
BlogEngine	>46,000	25.7	151	258	616	163
BugTracer	>500	10	19	298	223	93
NopCommerce	>10 Rel.	442	3781	5127	9110	484

Table 4. Analysis and mitigation results classification. **A**: analysis, **B**: mitigation

A	**Actuals**			**B**	**Actuals**		
		Vulnerability	N-Vulnerability			Blocked	N-Blocked
	Vulnerability	TP	FP		Blocked	TP	FP
	N-Vulnerability	FN	TN		N- Blocked	FN	TN

$$\textbf{\textit{Precision}} = \frac{TP}{TP+FP} \text{ (Eq. 1)} \qquad \textbf{\textit{Recall}} = \frac{TP}{TP+FN} \text{ (Eq.2)} \qquad \textbf{\textit{F-Measure}} = 2\,\frac{Precision*Recall}{Precision+Recall} \text{ (Eq. 3)}$$

Table 5. Experimental results of VAM-aaS using benchmark apps. **TP**: No. of true positives, **FP**: No. of false positives, and **FN**: No. of false negatives.

Component	App	SQLI			XSS			Authn. Bypass			Improper Authz.		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Analysis	Galactic	2	0	0	3	1	1	4	0	0	2	1	0
	Splendid	13	2	1	7	1	0	3	0	0	3	0	0
	KOOBOO	14	2	0	10	2	0	4	1	0	11	2	1
	BlogEngine	3	0	1	3	0	1	0	0	0	4	0	0
	BugTracer	9	0	1	0	0	1	3	0	1	1	1	0
	NopCommerce	19	2	0	4	0	1	0	0	0	0	0	1
	Webgoat	8	0	1	5	1	0	3	0	1	3	0	0
Total		**68**	**6**	**4**	**32**	**5**	**4**	**17**	**1**	**2**	**24**	**4**	**2**
Mitigation	Galactic	2	1	0	4	1	0	4	0	0	2	1	0
	Splendid	14	0	0	7	1	0	3	2	0	3	0	0
	KOOBOO	14	2	0	10	3	0	4	1	0	12	0	0
	BlogEngine	4	0	0	4	2	0	0	0	0	4	2	0
	BugTracer	10	0	0	1	0	0	4	1	0	1	1	0
	NopCommerce	19	0	0	5	0	0	0	0	0	1	0	0
	Webgoat	9	0	0	5	1	0	4	2	0	3	1	0
Total		**72**	**3**	**0**	**36**	**8**	**0**	**19**	**8**	**0**	**30**	**5**	**0**

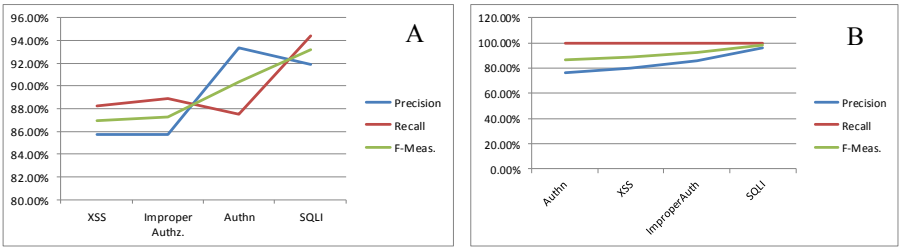

Fig. 6. Precision, Recall, and F-measure metrics of the analysis and mitigation components

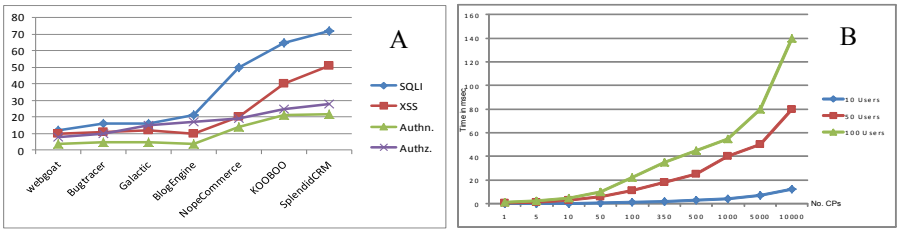

Fig. 7. Performance of vulnerability analyzer (per vulnerability type), and mitigation components

A high precision means that the approach gives more valid results (TP) than invalid results (FP). Thus, the maximum precision is achieved when no false positives given (see Equation 1). The recall metric is used to assess the completeness. A high recall means more valid results (TP) than missed valid results (FN), see Equation 2. The F-measure metric combines both precision and recall. It is used to measure the overall effectiveness of the approach (weighted harmonic mean). It depends on the importance of the recall rate and the precision rate e.g. if we are interested in high precision (more valid results) then we give precision factor high weight, and vice-versa. In our evaluation, we assume both are equally important, see Equation 3. In Table 5, we summarize our experiments' results applied on benchmark suite to identify four of the Top10 web applications vulnerabilities (OWSAP2010 report). For each component, we count number of TP, FP, and FN.

Fig. 6-A shows precision, recall, and F-measure for vulnerability analysis component on different vulnerability types using the benchmark applications. The average precision of the analysis component is around (90%). This means that in every identified (100) vulnerability instances, we have (10) false positives. Its recall is around (92%). This means that in every 100 vulnerabilities, we correctly identify 92 and miss 8. This can be improved by specifying more sound vulnerability signatures. Fig. 6-B, shows the precision, recall and F-measure for the mitigation component, we have incorporated the results returned from the analysis component with the FN missed vulnerabilities. It shows that the recall is (100%) which means that we did not miss any of the specified mitigations; however, we have an average of (85%) precision – i.e. high FP - as we may secure entities that have no security problem. This depends on the accuracy of the specified mitigation actions.

Fig. 7-A shows time (in sec) took in analyzing benchmark applications to pinpoint existing vulnerabilities. The SQLI vulnerability takes much more time to identify than XSS and authorization bypassing. The authentication bypass takes the lowest time. The time required to identify vulnerability instance depends on the complexity of the specified OCL signature. Fig. 7-B shows performance overhead of the mitigation component with different numbers of concurrent users and numbers of critical or vulnerable points - CPs (on a Core2Duo desktop PC with 4GB Memory). Overhead is equal to time spent by the security kernel to query security specification document to get security controls to be employed in securing intercepted point, and time spent in calling these security controls. Time spent by the security controls themselves we do not factor in, as this needs to be spent whether using our approach or traditional hard-coded security. Performance can be further improved using replicas of the mitigation component with different services and platforms.

7 Discussion

We developed a formal vulnerability definition schema including vulnerability signature and mitigation actions, extensible vulnerability analyzer based on the proposed signature specification schema and applied on abstract program representation, and a vulnerability mitigation approach based on dynamic and runtime injection of security controls into vulnerable entities. Use of OCL allowed us to make use of existing validation and query parsing tools. Use of this abstract representation helped us to generalize our analysis away from programming language and platform details. It also helps make our approach scalable for larger applications. Use of a common security interface allows integration of different security controls without a need to develop new system-security control connectors.

From our experience in developing signatures of the Top10 vulnerabilities and our experiments we determined that: **(i)** it is better to use dynamic analysis tools with certain vulnerabilities, such as Cross site reference forgery (CSRF), because these vulnerabilities can be handled by the web server. This means static analysis may result in high FP, if used; **(ii)** some vulnerabilities can be easily identified and located by static analysis such as SQLI and XSS vulnerabilities; **(iii)** some vulnerabilities such as DOM-based SQL and XSS vulnerabilities need a collaborating static and

dynamic analysis to locate them. We believe that combining static and dynamic analysis is needed to increase the precision and recall rates.

Our vulnerability analyzer achieves a precision rate of 90% and recall rate of 92%. These figures are actually impacted by the accuracy of the specified vulnerability signatures. Static analysis approaches usually result in high false positives as they work on source code level – i.e. the vulnerability may be addressed on the component or the application level. This problem can be solved by employing dynamic vulnerability analysis. However, dynamic vulnerability analysis approaches cannot help locating specific code snippets where vulnerabilities exist. Moreover, they do not help testing code coverage by generating all possible test cases.

From our experiments in the mitigation actions and security controls integrations, we found that although the use of web application firewalls is a straight forward solution, it is not always feasible to use WAF to block all discovered vulnerabilities. The selection of the entity level to apply security controls on (application, component, method, etc.) impacts the application performance – i.e. instead of securing only vulnerable methods, we intercept and secure (add more calls) the whole component requests. A key point that worth mentioning is that the administration of security controls should be managed by the service or cloud provider admins. We focus on integrating controls within vulnerable entities.

Our vulnerability mitigation component works online without a need for manual integration with applications and services under management. The overhead added by the mitigation action can be reduced if service developers add a new service patch. In this case, the vulnerability analysis component will not report a vulnerability. Thus, the mitigation component will not inject security controls.

8 Related Work

We are aware of no existing efforts that introduce an integrated solution to the vulnerability analysis and mitigation problem. Most focus on either vulnerability analysis or vulnerability mitigation, although vulnerability mitigation seems of less interest to date. Existing vulnerability analysis efforts can be categorized in static analysis-based, dynamic analysis, and hybrid approaches. Broadly, static analysis techniques work on source code level while dynamic analysis works on application as black-box.

NIST [18] has been running a security analysis tools assessment project (SAMATE). A part of this project is to specify a set of functional requirements that any source code security analysis approach should support. These include a set of weaknesses that an analyzer should be capable of identifying, including SQL injection, XSS, OS command injection, etc. They have also developed a set of test cases that help in assessing the capabilities of a security analysis tool in discovering such vulnerabilities. Halfond *et al.* [10] introduce a new SQL injection vulnerability identification technique base on positive tainting. They identify "trusted" strings in an application and only these trusted strings to be used to create certain parts of an SQL query, such as keywords or operators. Martin et al [7, 8] introduce a program query language PQL that can be used to capture definition of program queries that are capable to identify security errors or vulnerabilities. A PQL query is a pattern to be matched on execution traces. They focus on Java-based applications and define signatures in terms of code snippets. This limits their capabilities in locating vulnerabilities' instances that matches semantically but not syntactically. Wassermann *et al.*

[12] introduce an approach to finding XSS based on formalizing security policies using W3C recommendation. They conduct a string-taint analysis using CFG to represent sets of possible string values these are enforced on web pages to assure no untrusted scripts. Ganesh et al [15] introduce string constraint solver to check if a given string have a sub-string with a given set of constraints. They used it in white box SQLI testing.

Kals et al [3] introduce a vulnerability scanner that uses a black-box approach to scan web sites for the presence of exploitable SQLI and XSS vulnerabilities. They do not depend on a vulnerability signature database, but they require attacks to be implemented as classes that satisfy certain interfaces. Felmetsger et al [4] use an approach for automated logic vulnerabilities detection in web applications. They depend on inferring system specifications of a web application's logic by analysing system execution traces. They then use model checking to identify specification violations; however, they assume that collected traces represent real correct system behaviour.

Existing efforts for vulnerability mitigation can be categorized as code modification guidelines, server-side mitigation, and client side (Browser) approaches. Wurzinger et al [19] introduce SWAP as a server-side solution for detecting and preventing XSS. SWAP works as a reverse proxy intercepting HTML responses and forward them to a java-script detection component. Bisht et al [20] introduce a new XSS prevention solution that works by dynamically learning the set of scripts that a web application intends to create for any HTML request. Ravi et al [21] introduce an analysis of two mitigation approaches for XSS. The first one is a server-side approach intercept requests and perform string analysis looking for XSS signatures. The second approach is browser-based that replaces java related keywords in the response body with other words that have same pronunciation. Vogt et al [22] introduce a new approach to block XSS attacks on the client side by tracking the flow of sensitive information inside the web browser. Brunil et al [23] investigates security vulnerabilities and mitigation strategies to help developers build secure applications.

9 Summary

We described a new integrated, automated online vulnerability analysis and mitigation solution as a service (VAM-aaS). We use a formalized vulnerability definition schema including vulnerability signature and mitigation actions. An OCL-based vulnerability signature specifies a set of invariants that verifies the existence or absence of a given vulnerability in a target program. We developed a static vulnerability analysis tool that uses these signatures to locate possible matches in a target system. Vulnerability mitigation actions specify what to do whenever a vulnerability instance is reported, including security controls to be plugged-in to block discovered vulnerabilities. We validated our approach on a set of seven open source web applications. Our experimental results show that the OCL-based analysis tool achieves (90%) precision rate and (92%) recall rate while our mitigation component achieves (100%) recall and (85%) precision rate.

References

1. Almorsy, M., J. Grundy, and I. Mueller. *An analysis of the cloud computing security problem*. in *Proc. Asia Pacific Cloud Workshop, APSEC*. 2010. Sydney, Australia.
2. Bau, J., et al. *State of the Art: Automated Black-Box Web Application Vulnerability Testing*. in *Proc. of 2010 IEEE Symposium on Security and Privacy*. 2010.
3. Kals, S., et al., *SecuBat: a web vulnerability scanner*, in *Proc. of 15th Int. Conf. on World Wide Web*2006, ACM: Edinburgh, Scotland. p. 247-256.

4. Felmetsger, V., et al. *Toward automated detection of logic vulnerabilities in web applications*. in *Proc. 19th USENIX Conf. on Security*. 2010. Washington, DC.
5. Jovanovic, N., Kruegel C., and Kirda E.. *Pixy: a static analysis tool for detecting Web application vulnerabilities*. in *Proc. IEEE Symposium on Security and Privacy*. 2006.
6. Dasgupta, A., V. Narasayya, and M. Syamala. *A Static Analysis Framework for Database Applications*. in *Proc. of 2009 IEEE Int. Conf. on Data Engineering*. 2009.
7. Martin, M., B. Livshits, and M.S. Lam. *Finding application errors and security flaws using PQL: a program query language*. in *Proc. 20th Conf. on Object-oriented programming, systems, languages, and applications* 2005. CA, USA.
8. Lam, M.S., et al. *Securing web applications with static and dynamic information flow tracking*. in *Proc. of 2008 symposium on Partial evaluation and semantics-based program manipulation*. 2008. California, USA.
9. Kieyzun, A., et al. *Automatic creation of SQL Injection and cross-site scripting attacks*. in *Proc. of 31st Int. Conf. on Software Engineering*. 2009.
10. Halfond, W.G.J., A. Orso, and P. Manolios. *Using positive tainting and syntax-aware evaluation to counter SQL injection attacks*. in *Proc. of 14th Int. symposium on Foundations of software engineering*. 2006. Oregon, USA.
11. Weinberger, J., et al. *A systematic analysis of XSS sanitization in web application frameworks*. *Proc. 16th European Conf. Research in computer security*, 2011. Belgium.
12. Wassermann, G. and Z. Su. *Static detection of cross-site scripting vulnerabilities*. in *Proc. of 30th Int. Conf. on Software engineering*. 2008. Leipzig, Germany: ACM.
13. Hooimeijer, P., et al., *Fast and precise sanitizer analysis with BEK*, in *Proc. of 20th USENIX Conf. on Security*2011, USENIX Association: San Francisco, CA. p. 1-1.
14. Balzarotti, D., et al. *Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications*. in *Proc. IEEE S&P,* 2008.
15. Ganesh, V., et al. *HAMPI: a string solver for testing, analysis and vulnerability detection*. in *Proc. 23rd Int. Conf. on Computer aided verification*, 2011, Snowbird.
16. Monga, M., R. Paleari, and E. Passerini, *A hybrid analysis framework for detecting web application vulnerabilities*, in *Proc. of 2009 ICSE Workshop on Software Engineering for Secure Systems*2009, p. 25-32.
17. Zhang, R., et al., *Static program analysis assisted dynamic taint tracking for software vulnerability discovery.* Computers & Mathematics with Application, 2012. p. 469-480.
18. NIST, *Source Code Security Analysis Tool Functional Specification Version 1.1*, in *NIST Special Publication 500-268*May 2007, Accessed 2011.
19. Wurzinger, P., et al., *SWAP: mitigating XSS attacks using a reverse proxy*, in *Prof. ICSE Workshop on Software Engineering for Secure Systems,* 2009: Vancouver, p. pp. 33–39.
20. Bisht, P. and Venkatakrishnan V., *XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks*, in *Detection of Intrusions and Malware, and Vulnerability Assessment*2008, Springer Berlin / Heidelberg. p. 23-43.
21. Kotha, R.K., G. Prasad, and D. Naik, *Analysis of XSS attack Mitigation techniques based on Platforms and Browsers.* SEA, CLOUD, DKMP, CS & IT, 2012. **5**: p. 395–405.
22. Vogt, P., et al., *Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis* in *Network and Distributed System Security Symposium*2007: San Diego, CA.
23. Dalila, B., et al, *Security Vulnerabilities and Mitigation Strategies for Application Development*, in *Proc. 6th Int. Conf. on ITNG,* 2009. p. 235-240.
24. CENZIC. *Web Applications Security Trends Reports Q1-Q2 2010*, URL: www2.cenzic.com/downloads/Cenzic_AppSecTrends_Q1-Q2-2010.pdf
25. OWASP. *Open Web Application Security Project*, URL: https://www.owasp.org.
26. CWE. *Common Weaknesses Enumeration*, URL: http://cwe.mitre.org
27. SharpDevelop. http://wiki.sharpdevelop.net/
28. Yiihaw. *YIIHAW Is an Intelligent and High-performing Aspect Weave*, URL: http://yiihaw.tigris.org/