Detecting Inter-Component Vulnerabilities in Event-based Systems

Youn Kyu Lee
Multimedia Processing Lab
Samsung Advanced Institute of Technology, Suwon, Republic of Korea

***Abstract*—Event-based system (EBS) has become popular because of its high flexibility, scalability, and adaptability. These advantages are enabled by its communication mechanism—implicit invocation and implicit concurrency between components. The communication mechanism is based on non-determinism in event processing, which can introduce inherent security vulnerabilities into a system referred to as event attacks. Event attack is a particular type of attack that can abuse, incapacitate, and damage a target system by exploiting the system's event-based communication model. It is hard to prevent event attacks because they are administered in a way that does not differ from ordinary event-based communication in general. While a number of techniques have focused on security threats in EBS, they do not appropriately resolve the event attack issues or suffer from inaccuracy in detecting and preventing event attacks. To address the risk of event attacks, I present a novel vulnerability detection technique for EBSs that are implemented by using message-oriented middleware platform. My technique has been evaluated on 25 open-source benchmark apps and eight real-world EBSs. The evaluation exhibited my technique's higher accuracy in detecting vulnerabilities on event attacks than existing techniques as well as its applicability to real-world EBSs.**

Keywords*—Event-based system; program analysis; software security

I. Introduction

Event-based systems (EBSs) implemented by using MOM platforms are widely used. They are implemented in various types of systems such as web apps or SOA-based systems by using different types of MOM platforms such as Prism-MW [1], Java Message Service [3], and Siena [10]. EBSs have become popular because of its high flexibility, scalability, and adaptability. These advantages are enabled by its reliance on implicit invocation and implicit concurrency. Specifically, in EBSs, components may not know the consumers of the events they publish, nor do they necessarily know the producers of events they consume. However, this communication mechanism is based on non-determinism in event processing, which can introduce inherent security vulnerabilities into a system referred to as event attacks. For example, developers may build EBSs by utilizing externally developed components that contain malicious code, and users may use those EBSs comprising malicious components. For those cases, malicious components can launch unintended behaviors through event communication, such as eavesdropping on events to steal sensitive information or exploiting the information in events to hijack the system's functionalities.

Existing system analysis techniques neither focus on event attacks nor correctly detect vulnerabilities across components [5,6,7,23]. Specifically, existing vulnerable-flow analysis techniques do not support implicit invocation between components and are not scalable to analyzing systems comprising large numbers of components [6,7,12]. While a large body of research has studied detecting vulnerabilities that expose Android apps to event attacks [9,11,12,13], they cannot be directly applied to other types of EBSs, because Android uses its system-specific communication model, APIs, and component life-cycles. Thus a generalized solution is required to protect other types of EBSs.

To overcome aforementioned challenges and the shortcomings of the existing approaches, I designed a technique that automatically detects target EBS's vulnerabilities that expose the system to event attacks. My solution statically inspects target EBS in order to identify security vulnerabilities that expose the system to event attacks. It performs vulnerable-flow analysis and pattern matching on event communication channels between components. My technique is distinguished from prior works because (1) it detects potential risks of event attack in EBSs more accurately than existing techniques, (2) it supports multiple types of MOM platform, and (3) it enables a scalable analysis of EBSs comprising a large number of components and methods.

This paper makes the following contributions: (1) I proposed a novel technique that identifies security vulnerabilities from multiple types of EBSs; (2) I developed a prototype tool that implements the proposed technique; (3) I provided the results of evaluations that involve real-world EBSs and comparable techniques. Section 2 illustrates event attacks in EBSs, which motivate my research. Section 3 details my approach and Section 4 presents the evaluations of my technique. A discussion of related work is provided in Section 5, and my conclusions are presented in Section 6.

II. Motivating Example: Web Applications

In this section, I will present a simplified example of event attack which can be launched on event-based web apps. Fig. 1 and 2 illustrate eavesdropping attack. An app App1 follows event-based communication model and is implemented by using Java Message Service [3], a Java MOM platform for message-based communication between components. App1 is corrupted to contain an unintended component Mal (in Fig. 2) so that event attacks can be launched. Fig. 1 and 2 show where App1's vulnerability resides. In this app, all events are published through "CustomTopic".

```
1  public class Vic {
2   ...
3  String s = getSensitiveInfo();
4  Topic topic = (Topic)ctx.lookup("CustomTopic");
5  TopicConnection con = factory.createTopicConnection();
6  TopicSession session = con.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
7  TopicPublisher publisher = session.createPublisher(topic);
8  Message e1 = session.createMessage();
9   e1.setJMSType("TextMessage");
10  e1.setName("ReplyInfo");
11  e1.setStringProperty("Sensitive", s);
12  publisher.publish(e1);
13  }
```

Fig. 1. Component Vic in App1.

```
1   public class Mal {
2   ...
3   String m;
4   Topic topic = (Topic)ctx.lookup("CustomTopic");
5   TopicConnection con = factory.createTopicConnection();
6   TopicSession session = con.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
7   TopicSubscriber subscriber = session.createSubscriber(topic);
8   subscriber.setMessageListener(new MessageListener(){
9    protected void handleMessage(Message e2){
10  if (e2.getName().equals("ReplyInfo")){
11     m = e2.getStringProperty("Sensitive");
12  }}}
```

Fig. 2. Component Mal in App1.

Component Vic in App1 (in Fig. 1) publishes an event e1 through CustomTopic without any particular protection such as access restrictions. e1 has two attributes— one with the name “Name” (whose value is “ReplyInfo”) and one with the name “StringProperty” (whose value is “Sensitive”)—while containing sensitive information (i.e., s). By listening to “CustomTopic” and declaring attributes “ReplyInfo” and “Sensitive”, Mal can eavesdrop on the event sent from Vic and obtain the sensitive information.

As shown in this example, since event attacks appear to be ordinary event interactions, existing malware inspection techniques, especially the techniques that rely on signature-based detection [23], may not be able to detect event attacks. Moreover, since publishing and consuming events can be processed via ambiguous interfaces, existing flow-analysis techniques will be unable to accurately analyze implicit invocation between components. Furthermore, since routing event is performed in an invisible and non-deterministic way, it is difficult to expect when and where the event attacks are actually launched.

III. Solution

My proposed solution basically considers three main challenges as follows: (1) ambiguous event communication channels: EBS’s inherent attributes hamper the extraction of event communication channels via which events are exchanged between components. Specifically, implicit invocation between components makes it difficult to determine where each event will flow into, and EBS’s event interfaces do not explicitly reveal the events to be consumed. Furthermore, depending on the types of MOM platform, different event interfaces can be used. To handle this, my technique leverages Eos [4], a technique that statically extracts event types and their attributes based on the characteristics of underlying MOM platform; (2) scalable flow analysis: To check whether sensitive data leaks or unintended access to sensitive functionality can be launched, control-/data-flow analysis on methods in each component is required. However, in case when an EBS comprises a large number of components and methods, flow analysis on every method in the EBS may not be scalable. According to prior research. [8], on average, EBSs contain over 35 methods to be analyzed, which could consume hours for a real-world EBS. Although several flow-analysis techniques have been proposed for Android apps [12,19], considering the fact that mobile platforms limit the size of apps, those techniques may not scale with large-scale EBSs containing methods with larger size and higher complexity. My technique provides a size reduction algorithm which enables its analysis to scale well with identifying vulnerabilities from large-scale EBSs; (3) inconstant distinction of components. Event attacks are launched across the components that have different trust level. Although Android uses a consistent mechanism for distinguishing among the trust levels of app components (i.e., each “app” has different trust level), other EBSs may use different types of distinction depending on their system configuration. For example, the trust level of externally-developed components can be different from that of component developed in-house. To handle this, my technique introduces the concept of trust boundaries. A trust boundary is defined as a unit for dividing components based on each component’s

trust level. Components that have the same trust level belong to the same trust boundary, and a trust boundary can be set per each component as well as a group of components.

My solution operates in three phases—Extraction, Reduction, and Identification—and uses three types of inputs: the target EBS's (1) implementation, (2) configuration, and (3) sensitive APIs. The configuration includes the information regarding the underlying MOM platform (i.e., the methods for event communication and the base class for events) and trust boundaries. The information of underlying MOM platform can be derived from the API specification of the platform, which only needs to be identified once per platform. Considering the existing platforms, such information has been publicly accessible. Trust boundaries can be easily derived by clustering components based on a developer's trust level regarding each component. While a set of sensitive APIs relies on the expectation that developers can provide accurately, it is fairly straightforward to identify them. Because so far as the components developed in-house are concerned, they might know particular APIs that handle important data or sensitive functionalities. Furthermore, even if a developer is not fully knowledgeable about the sensitive APIs in the target system, she can refer to the existing sets of APIs [2] which are generally considered as sensitive. According to the results of evaluation in Section 4, relying on setter and getter methods which are generally considered as sensitive, indicated a fairly high precision (=85.67%) in identifying vulnerabilities. In the remainder of this section, I will discuss each of three phases in detail.

Extraction - In this phase, target system's implementation is inspected in order to extract two different information: (1) The first information includes published event types (PET) and consumed event types (CET) accessed by each component, which can be used to infer event communication channels between components [8]. By using static flow-analysis on the target system's implementation, every component's PET and CET are extracted along with corresponding attributes from the system implementation. In Fig. 1, an example of PET published at line 12 is {(Name: "ReplyInfo"), (StringProperty: "Sensitive")}; (2) The Second information is the location where each sensitive API is accessed or called. For each method in a given list of sensitive APIs, the components where the method is called are identified along with their location in the system implementation.

Reduction-To identify vulnerable event communication channels, both inter- and intra-component flows are considered by combining the extracted event types with each component's control-flow graph (CFG). However considering a large-scale EBS, it may not be scalable to generate and traverse every component's CFG. To address this, we build an event flow graph (EFG), which provides a macro perspective of target EBS (see Fig. 3), and examines the EFG in order to prune the components that are unnecessary for subsequent analyses.

In an EFG, components are connected by the edges that represent event communication channels between pairs of components. An edge is determined by matching PET and CET, while having a direction to which an event is being sent. For the component where a sensitive API is called, my solution checks if its sensitive API is reachable from or to its event interfaces—consuming event interface (CEI) and publishing event interface (PEI)—via its call graph (CG). If yes, the component is labeled as a sensitive component (see Fig. 3). The components that form an event communication channel across trust boundaries are labeled as boundary components. If a boundary component's PEI for event communication across trust boundaries is reachable from its CEI or sensitive API via CG, its attribute is set to be outflow-boundary (OB). Conversely, if its CEI for event communication across trust boundaries is reachable to its PEI or sensitive API via CG, its attribute is set to be inflow-boundary (IB).

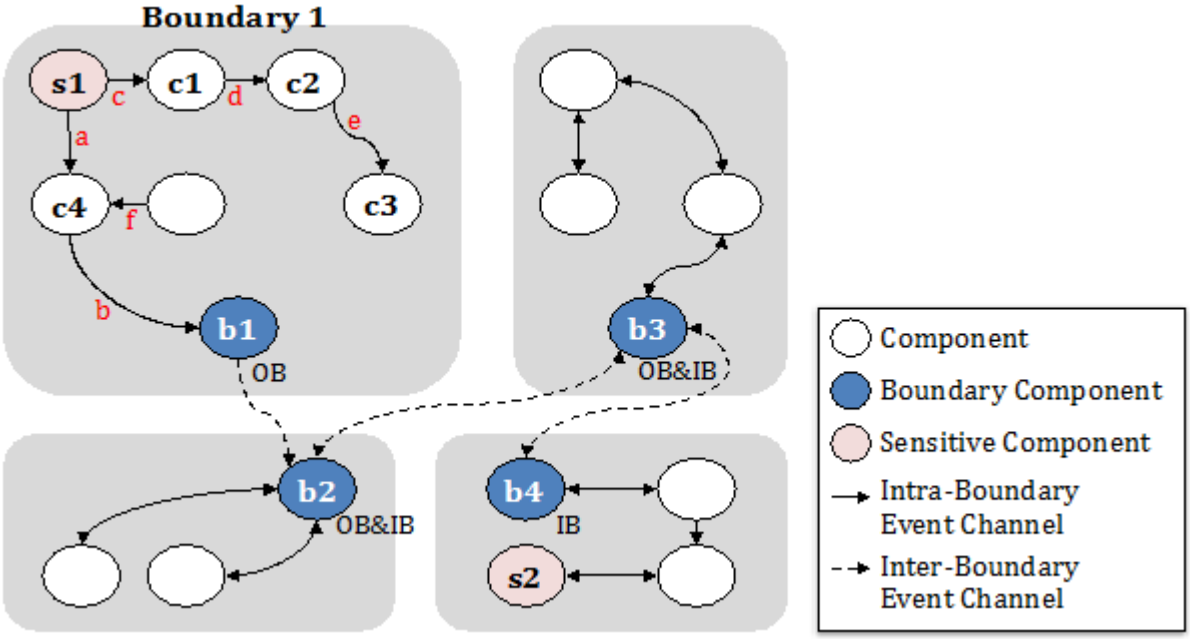

Fig. 3. An Event Flow Graph.

Algorithm 1. Identification of Vulnerable Communication Channels

Input: $G \Leftarrow$ an EFG
Output: $VulCF \Leftarrow$ a set of vulnerable event communication channels
1 Let S_G be a set of sensitive components in G
2 Let OB_G be a set of outflow-boundary components in G
3 Let IB_G be a set of inflow-boundary components in G
4 Let SM_c be a set of sensitive methods in a component $c \in G$
5 Let $l = [c1, c2, ... cn]$ be a list of connected components from component $c1 \in S_G$ to component $cn \in OB_G$ or from component $c1 \in IB_G$ to component $cn \in S_G$
6 Let $t \in (PET_c \cup CET_c)$ where $\forall c \in G$
7 **foreach** $l \in G$ **do**
8 **if** $((c1 \in S_G) \wedge (cn \in OB_G))$ **then**
9 **foreach** $s \in SM_{c1}$ **do**
10 $t \Leftarrow$ *identifyFlow*($c1$, s, PEI_{c1}, "out", l.remove($c1$))
11 add *getOutFlowChannel*(t) to *VulCF*
12 **if** $((cn \in S_G) \wedge (c1 \in IB_G))$ **then**
13 **foreach** $s \in SM_{cn}$ do
14 $t \Leftarrow$ *identifyFlow*(cn, s, CEI_{cn} , "in", l.remove(cn))
15 add *getInFlowChannel*(t) to *VulCF*
16 return *VulCF*

My solution prunes the components that are not associated with vulnerable event communication. For example, in Fig. 3, component s1 publishes two different types of events (i.e., a and c) each of which initiates different subsequent event communication (i.e., b and d-e, respectively). Considering the fact that event attacks exploit (1) event communication across trust boundaries and (2) event communication that flows into or from sensitive APIs, event communication channels for c, d, and e are not essentially vulnerable to event attacks, because they are not involved in the event communication across trust boundaries. Thus, the components that are connected with those event channels are removed (i.e., components c1, c2, and c3) from EFG in order to reduce the overhead in subsequent flow analyses.

Identification-Vulnerable event communication channels are identified by implementing Algorithm 1 on the pruned EFG. Algorithm 1 iterates over each list of connected components (i.e., l in G), which directs from a sensitive component to a boundary component or reverse (lines 7-15). Two cases are considered depending on the direction of l:

(1) For l which directs from a sensitive component to an outflow-boundary component (lines 8-11), Algorithm 1 checks if an intra-component flow exists between a sensitive methods and PEI of $c1$ (=the starting component of l) by calling *identifyFlow* with the flag as "out" (line 10). To illustrate this case, consider the component Vic in Fig. 1. Since Vic is a sensitive component and an out-flow boundary component, Algorithm 1 checks if an intra-component flow exists between its sensitive method *getSensitiveInfo* and its PEI publish by calling *identifyFlow*. *identifyFlow* checks if a given component contains an intra-component flow between given two methods (i.e., $m1$: s and $m2$: PEI_{c1}). In case when a given flag is "out", it inspects every node in the CFGs of $m1$ and $m2$, and checks if a node in $m2$ is dependent on a node in $m1$. If yes, it recursively checks an intra-component flow from CEIs to PEIs of subsequent components in l. If the flows exist throughout every component in l, it returns PET which can be published via $m2$; Otherwise it returns null. For the reverse case when flag is "in", it checks the flow from nodes in $m2$ to node in $m1$, and recursively identifies intra-component flows from PEIs to CEIs of subsequent components in l. If the flow exists through every component in l, it returns CET, which can be consumed via $m1$; Otherwise, it returns null. If *identifyFlow* returns PET (i.e., t) which is not null, Algorithm 1 identifies the event communication channel where the returned PET is published by calling *getOutFLowChannel*, and add the channel to *VulCF*, a set of vulnerable event communication channels (lines 10-11). Coming back to the example in Section 2, since Vic contains an intra-component flow from *getSensitiveInfo* to publish, the PET (i.e., {(Name: "ReplyInfo"), (StringProperty: "Sensitive")}) will be returned by Algorithm 1. Finally, the communication channel between Vic and Mal will be added to *VulCF*.

(2) The second case is for l which directs from an inflow-boundary component to a sensitive component (lines 12-15). Algorithm 1 checks if an intra-component flow exists between a sensitive method s and CEI of cn (=the last component of l) by calling *identifyFlow* with the flag as "in" (line 14). If *identifyFlow* returns CET (i.e., t) which is not null, Algorithm 1 identifies the event communication channel where the returned CET is consumed by calling *getInFlowChannel*, and add the channel to *VulCF* (line 15).

My solution also performs pattern analysis on the event communication channels in EFG based on the previously identified patterns [9]. Four different patterns are considered as follows: (c: a component, T: a trust boundary, $x \Rightarrow y$: an event communication channel exists from x to y).

(1) For components c1 and c2 $\in$ T1, c3 $\in$ T2; c1 = c2 = c3: $(c3 \Rightarrow c2) \wedge (c1 \Rightarrow c2)$
(2) For components c1 and c2 $\in$ T1, c3 $\in$ T2; c1 = c2 = c3: $(c1 \Rightarrow c3) \wedge (c1 \Rightarrow c2)$
(3) For components c1 $\in$ T1; c2 and c3 $\in$ T2; c1 = c2 = c3: $(c1 \Rightarrow c2) \wedge (c2 \Rightarrow c3) \wedge \neg(c1 \Rightarrow c3)$
(4) For components c1 and c2 $\in$ T1, c3 $\in$ T2; c1 = c2 = c3: $(c1 \Rightarrow c2) \wedge (c2 \Rightarrow c3) \wedge \neg(c1 \Rightarrow c3)$

The patterns are based on the assumption that event communication within the same trust boundary is intended access, but event communication across the boundaries can be unintended access from a malicious component. Specifically, in case of the pattern (1), c3 $\Rightarrow$ c2 can be spoofing. For the pattern (2), c1 $\Rightarrow$ c3 can be interception or eavesdropping. For the pattern (3) and (4), c1 $\Rightarrow$ c2 $\Rightarrow$ c3 can be confused deputy or collusion. If a given EFG contains event communication channels that match any of these patterns, the corresponding channel(s) to *VulCP* (i.e., a set for vulnerable event communication channels) are returned. Finally, all the identified event communication channels in *VulCF* and *VulCP* are returned. While the channels belonging to both sets can be considered as the most vulnerable, other ones also need to be inspected and protected in order to minimize the threats of event attacks in a target EBS.

IV. Evaluation

I have implemented the prototype of my solution as a stand-alone Java app which combines approximately 2,000 newly written SLOC with the off-the-shelf tools, Eos [4] and Soot [21]. Eos is used in the extraction phase to extract PET and CET from target EBS. Soot is used to generate CGs and CFGs of the components within a target EBS. The prototype was empirically evaluated in terms of its accuracy, applicability, and performance in detecting vulnerabilities from a target EBS's byte-code.

A. Accuracy

This evaluation targeted vulnerability detection tools for web apps, because they fall under a particular type of EBS which can be implemented by using MOM platforms. Among the state-of-the-art static analysis tools for detecting security vulnerabilities in web apps, three tools were executable while supporting Java-based systems: Xanitizer [7], Owasp Orizon [6], and SonarQube [5]. I evaluated my prototype's accuracy in identifying vulnerable event communication channels by comparing its results against those three tools.

Since existing test benchmarks for web apps neither target EBSs nor event attacks, I have created a test benchmark for evaluating security analysis techniques for EBSs. To minimize internal threats to the validity of results, I asked graduate students at USC to build a set of apps that implement event attacks based on the published literature [9]. They built 20 distinct event-based apps by using two representative types of MOM platforms (10 apps for each): (1) Java Message Service [3], the widely adopted Java-oriented middleware; and (2) Prism-MW [1], a research-off-the-shelf middleware platform for distributed software systems. Every app was designed to contain a malicious component that had the sole purpose of launching an event attack. The benchmark also comprises five "trick" apps containing vulnerable but unreachable components, whose identification would be a false warning. This yielded a total of 25 event-based apps containing 20 vulnerable event communication channels.

I ran the three tools on my test benchmark and measured their (1) precision, i.e., identified vulnerabilities that were actually vulnerable to event attacks, and (2) recall, i.e., the ratio of identified vulnerabilities to all those exposed to event attacks. My prototype detected vulnerable event communication channels with 100% precision and recall, correctly ignoring all "trick" cases. However, other tools (i.e., Xanitizer, Owasp Orizon, and SonarQube) were unable to find any of the vulnerabilities related to event attacks from the benchmark. Specifically, Xanitizer did not return any vulnerability. While Owasp Orizon and SonarQube reported some security warnings (e.g., potential dangerous keyword in the method), they are not directly related to the vulnerabilities caused by event attacks. This is primarily because these three tools neither target event attacks nor support inter-component flow analysis.

B. Applicability

To assess if my solution is applicable to real-world EBSs, I selected eight EBSs from the test suite which have been used in evaluating prior research [4]. While all subject systems are implemented in Java, they are from different app domains (e.g., game, simulator, and chat system), of different sizes (5K-247IK SLOC), and use different underlying mechanisms (e.g., JMS [3], Prism-MW [1], and REBECA [16]) for event communication. Since the list of sensitive APIs and trust boundaries were not provided for those systems, I have used the configuration that every 'getter' or 'setter' method was a sensitive method and every component belonged to different trust boundaries. According to the well-known sensitive API list for Android [18], 81% of sensitive methods are eight getters or setters (getter: 97%, setters: 65%), which implies that getters and setters are more likely to be sensitive to security attacks compared to other methods. However, it is important to note that this does not necessarily meant that all getters and setters are always sensitive methods. Among the eight subject systems, my prototype flagged 25 vulnerable event communication channels in three systems (Dradel: 12, ERS: 11, KLAX: 2). On average, the precision of result was 85.67% (Dradel: 75%, ERS: 82%, KLAX: 100%). Every false positive was caused by the prototype's inaccuracy in identifying control-flows between sensitive methods and event interfaces. For those three systems, Xanitizer reported 83 security warnings such as "may expose internal representation by returning reference to mutable object" and "IO Stream Resource Leaks" (Dradel: 6, ERS: 62, KLAX: 15). However only seven of them (8.43%) were related to the vulnerabilities that expose the system to event attacks. Owasp Orizon and SonarQube returned 13 (Dradel: 9, ERS: 1, KLAX: 3) and 95 (Dradel: 17, ERS: 73, KLAX: 5) implementation bugs, respectively, indicated as "empty catch detected" and "found potential dangerous keyword". But none of them were related to the vulnerabilities that expose the system to event attacks. Those three tools also did not return any such vulnerability from the other five subject systems. Although my prototype outperformed the three tools in this evaluation, it is to be noted that they detected additional types of vulnerabilities my prototype does not target.

I also tested my prototype on the event-based apps comprising different numbers of components. I created four distinct apps by adding different numbers of components (i.e., 25, 50, 75, 100, respectively) to an app randomly selected from my benchmark. To check the prototype's best-case performance overhead, each of the added components is designed to have a minimized architecture—containing one method for communicating with at most two other components (55 SLOC)—which would induce the shortest analysis time while connected with other components. The size of the apps spanned 2.8K-7K SLOC. None of the added components are involved in the vulnerable event communication channels so that they can be pruned in Reduction phase. Then I measured the analysis time for each app both "with" and "without" the Reduction phase. The result (see Fig. 4) indicates that as the number of added components increased, the difference of analysis time between "with" and "without" Reduction phase also increased. This result confirms that my solution minimizes the potential overheads in its analysis by introducing the pruning operation. Considering the fact that the added components are designed to have a minimized architecture, the effectiveness of pruning will drastically increase in the case of large-scale EBSs comprising a number of components with higher complexity.

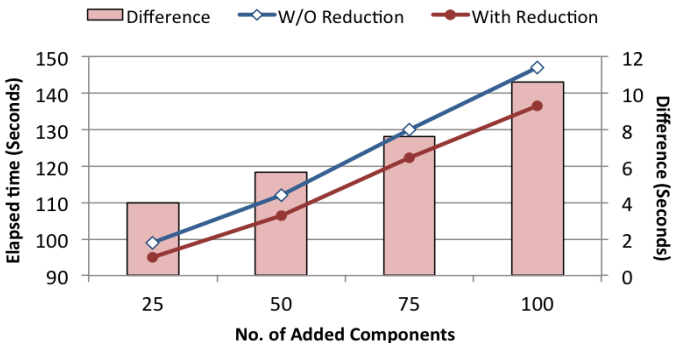

Fig. 4. Performance on different Number of Components.

V. Related Work

Several approaches have targeted the security in EBSs [15,17,20,22,25]. Simeon et al. [25] examined the security vulnerabilities of event-driven systems and defined the conditions that produce them. In general, existing security solutions for EBS employ encryption, static code analysis, and/or runtime access control techniques.

Encryption is widely used technique for securing not only general software systems, but also EBSs. EventGuard [22] proposes encryption for publish/subscribe systems in which each component encrypts events through event broker network. Publishers sign events and encrypt them with a random key, while the signature itself is encrypted with a topic-specific key and is attached to the event. However, encryption techniques increase the risk of compromised keys and may cause unacceptable performance overhead. Furthermore, key distribution is in appropriate when it is not determined which component will comprise the system.

Static code analysis is a popular technique for inspecting security flaws in target systems. SABER [14] is a static analysis tool that detects common design errors based on the instantiations of error pattern templates. Andromeda [27] inspects data-flow propagations on demand, while supporting apps written in Java, .NET, and JavaScript. Xanitizer [7] statically detects security vulnerabilities such as injections and privacy leaks by using taint-flow analysis. Owasp Orizon [6] is a source code security scanner designed to spot vulnerabilities in J2EE web apps by using pattern matching. SonarQube [5] is an open source platform for inspection of code quality to detect security vulnerabilities.

Runtime access control is another popular technique for securing EBSs. Alex et al. [24] proposed a policy model and framework for content-based publish/subscribe systems. DEFCon [26] is a middleware that applies an information flow control model which tracks the event flows through a complex, heterogeneous event processing system and constrains undesirable event flows that could potentially violate security policy. However, aforementioned techniques are more focused on other types of security issues than event attacks. Furthermore, since those techniques do not fully support event-based communication model, they may suffer from inaccuracy and scalability problems in analyzing large-scale web apps comprising a number of components.

VI. Conclusion

While event-based communication model enables highly decoupled, scalable, and easy-to-evolve systems, the non-determinism in event processing can be exploited by event attacks. Existing solutions for general software systems cannot be directly applied to resolve event attacks because they do not support event-based communication model. Furthermore, existing security solutions targeting EBSs do not appropriately resolve event attacks or suffer from inaccuracy in detecting event attacks.

To minimize the risk of event attacks, this paper presented a novel vulnerability detection technique for EBSs that are implemented by using MOM platforms. My technique statically analyzes vulnerabilities by examining inter-component flows and event communication patterns. It improves upon existing techniques in detecting vulnerabilities that expose the system to event attacks from a given EBS, while supporting multiple types of MOM platforms and increasing the coverage, accuracy, and scalability of vulnerability detection. My empirical evaluation demonstrates that my technique is more accurate in identifying vulnerable event communication channels from 33 EBSs compared to the state-of-the-art vulnerability detection techniques for web apps. The result of performance analysis shows that my technique is scalable to large-scale EBSs.

Future studies can focus on building a runtime-access controller which controls runtime event communication based on the statically-analyzed vulnerabilities. Also I can apply a visualization technique which can display the identified vulnerabilities between components in order to help engineer's understanding.

References

[1] Prism-MW-Architectural Middleware for Mobile and Embedded Systems. http://sunset.usc.edu/~softarch/Prism/, 2001.

[2] Which methods should be considered “Sources”, “Sinks” or “Sanitization”? http://thecodemaster.net/methods-considered-sources-sinks-sanitization/, 2015.

[3] Java Message Service (JMS). http://www.oracle.com/technetwork/java/jms/index.html, 2016.

[4] Automated Recovery of Software System Designs. https://softarch.usc.edu/projects/automated-recovery-of-software-system-designs/, 2016

[5] Continuous Code Quality | SonarQube. https://www.sonarqube.org/, 2017.

[6] Owasp Orizon. https://www.owasp.org/index.php/Category:OWASP_Orizon_Project, 2017.

[7] Xanitizer. https://www.rigs-it.net/index.php/product.html, 2017.

[8] DEvA, http://www-scf.usc.edu/~gsafi/FSE2015Replication/, 2015.

[9] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys), pages 239–252, 2011.

[10] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. 2001. Design and evaluation of a wide-area event notification service. ACM Trans. Comput. Syst. 19, 3, pages 332-383, 2001.

[11] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android Taint Flow Analysis for App Sets. In Proceedings of the 3rd International Workshop on the State of the Art in Java Program Analysis (SOAP), pages 1–6, 2014.

[12] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android App. In Proceedings of the 37th International Conference on Software Engineering (ICSE), pages 280–291, 2015.

[13] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), pages 229–240, 2012.

[14] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. SABER: Smart Analysis Based Error Reduction. In Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pages 243–251, 2004.

[15] Leonardo Aniello, Roberto Baldoni, Claudio Ciccotelli, Giuseppe Antonio Di Luna, Francesco Frontali, and Leonardo Querzoni. The Overlay Scan Attack: Inferring Topologies of Distributed Pub/Sub Systems Through Broker Saturation. In Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS), pages 107–117, 2014.

[16] G. Mühl et al. Distributed Event-Based Systems. Springer-Verlag New York, Inc., 2006.

[17] Fabio Petroni, Leonardo Querzoni, Roberto Beraldi, and Mario Paolucci. Exploiting User Feedback for Online Filtering in Event-based Systems. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC), pages 2021–2026, 2016.

[18] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A Machine-Learning Approach for Classifying and Categorizing Android Sources and Sinks. In Proceedings of the 21st Annual Network & Distributed System Security Symposium (NDSS), 2014.

[19] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), pages 259–269, 2014.

[20] Brian Shand, Peter Pietzuch, Ioannis Papagiannis, Ken Moody, Matteo Migliavacca, David Eyers, and Jean Bacon. Security Policy and Information Sharing in Distributed Event-Based Systems. Reasoning in Event-Based Distributed Systems, pages 151–172, 2011.

[21] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), pages 13–, 1999.

[22] Mudhakar Srivatsa, Ling Liu, and Arun Iyengar. EventGuard: A System Architecture for Securing Publish-Subscribe Networks. ACM Transactions on Computer Systems (TOCS), 29(4):10:1–10:40, December 2011.

[23] Handong Wu, Stephen Schwab, and Robert Lom Peckham. Signature based network intrusion detection system and method, September 2008. US Patent 7,424,744.

[24] Alex Wun and Hans-Arno Jacobsen. A Policy Management Framework for Content-based Publish/Subscribe Middleware. In Proceedings of the ACM/IFIP/USENIX 1907 International Conference on Middleware (Middleware), pages 368–388, 2007.

[25] Simeon Xenitellis. Security Vulnerabilities in Event-Driven Systems. In Proceedings of the IFIP TC11 17th International Conference on Information Security: Visions and Perspectives (SEC), pages 147–160, 2002.

[26] Peter Pietzuch. Building Secure Event Processing Applications. In Proceedings of the First International Workshop on Algorithms and Models for Distributed Event Processing (AlMoDEP), pages 11–11, 2011.

[27] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. ANDROMEDA:Accurate and Scalable Security Analysis of Web Applications. In Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE), pages 210–225, 2013.