A Grey-Box Approach for Detecting Malicious User Interactions in Web Applications

Wafa Ben Jaballah
Orange Labs, France
wafa.benjaballah@orange.com

Nizar Kheir
Thales Group, France
nizar.kheir@thalesgroup.com

ABSTRACT

Web applications are the core enabler for most Internet services today. Their standard interfaces allow them to be composed together in different ways in order to support different service workflows. While the modular composition of applications has considerably simplified the provisioning of new Internet services, it has also added new security challenges; the impact of a security breach propagating through the chain far beyond the vulnerable application. To secure web applications, two distinct approaches have been commonly used in the literature. First, white-box approaches leverage the source code in order to detect and fix unintended flaws. Although they cover well the intrinsic flaws within each application, they can barely leverage logic flaws that arise when connecting multiple applications within the same service. On the other hand, black-box approaches analyze the workflow of a service through a set of user interactions, while assuming only little information about its embedded applications. These approaches may have a better coverage, but suffer from a high false positives rate. So far, to the best of our knowledge, there is not yet a single solution that combines both approaches into a common framework.

In this paper, we present a new grey-box approach that leverages the advantages of both white-box and black-box. The core component of our system is a semi-supervised learning framework that first learns the nominal behavior of the service using a set of elementary user interactions, and then prune this nominal behavior from attacks that may have occurred during the learning phase. To do so, we leverage a graph-based representation of known attack scenarios that is built using a white-box approach. We demonstrate in this paper the use of our system through a practical use case, including real world attack scenarios that we were able to detect and qualify using our approach.

Keywords

Black-box; White-box; Web Security

MIST'16, October 28 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4571-2/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2995959.2995966

1. INTRODUCTION

Web applications are omnipresent in our daily life. They supply a wide range of services, spanning from social networks and communication services, to e-commerce, banking, and healthcare [1, 2, 3, 4, 5]. The modular composition of web applications facilitates greatly the provisioning of new Internet services, thanks to their ease of integration and adaptability. They can be dynamically composed together through standard API interfaces, moving a target service from a two-tier model, including the user and the service provider, towards a multi-tier model involving multiple application providers that interface together in order to supply a service. However, the multi-tier service provisioning model adds new bottlenecks in terms of security, because it would be difficult to guarantee a common level of security across all the applications that constitute a given service. In particular, the composition of elementary applications may add new security flaws that cannot be observed at the level of each single application [6].

To test internet services against any potential vulnerabilities, researchers have proposed a large number of techniques such as active pentesting, code analysis, and vulnerability scanners [7, 8, 9, 10, 11]. Today, multiple security solutions implement white-box techniques, mostly using static code analysis and vulnerability scanners that aim at detecting security flaws and well-defined vulnerability patterns. However, more subtle vulnerabilities that are specific to the workflow of a service, may arise after connecting together different single applications. These vulnerabilities are still discovered through error prone manual inspection, leaving a large number of flaws undetected. Yet best practice security solutions, such as educating web developers, although they may increase awareness, they cannot guarantee an acceptable level of security.

To handle these limitations, black-box techniques offer an alternative approach that leverages security flaws intrinsic to a service workflow [4, 12]. They aim at establishing a baseline model for a service, using either passive learning or active learning, and then to analyze this model against a set of known threat scenarios. Active fuzzing consists of testing a set of possible user inputs, and observing the outputs of the service to compare against its nominal behavior [13]. Any sequence of inputs that may drive the service into an inconsistent state is assimilated to a security flaw [7, 12]. On the other hand, passive learning applies statistical models and machine learning in order to characterize the nominal behavior of a service using a set of input observations, such as access logs or elementary user requests [4].

While black-box techniques complement the white-box approach, they still suffer from multiple limitations. First, active fuzzing techniques usually have a limited coverage, as they may only detect attacks that were considered in the fuzzing phase. Moreover, passive learning techniques usually have a high false positives rate, yet they are unable to handle attacks that may occur during the learning phase. These would be considered as benign by the system, and so they may also lead to false negatives during detection.

To the best of our knowledge, there is not yet a comprehensive approach that efficiently leverages the advantages of both white-box and black-box mechanisms.

The major technical barriers come from the fact that passive learning considers the application as a sealed machine with unobservable internals, and so it may only leverage external interactions with the service. White-box techniques, however, leverage the source code of an application; but they do not account for service workflows, nor do they analyze the way users interact with the service. Hence, white-box techniques are less efficient in detecting security flaws that occur when composing multiple elementary applications. Yet black-box techniques still lack the appropriate models and formalism that make possible to handle elementary flaws discovered using a white-box technique. They are unable to leverage these flaws during the learning process in order to prune the behavioral model and eliminate attacks that may have happened during the learning phase.

In this paper, we introduce a new approach, along with the appropriate models and formalism that address the aforementioned problems. To do so, we first propose a black-box approach that leverages a set of real world user interactions with the service, and further applies unsupervised clustering in order to infer a behavioral model for the service. This model is represented through a behavior graph where nodes represent elementary user requests and edges represent logical transitions between requests occuring within the same session with the service. Then, our system leverages information about possible security flaws that may have been detected using standard white-box techniques. These flaws are represented as attack graphs where nodes capture possible vulnerabilities and edges capture attack paths through the target service. Our system uses the information in the attack graph in order to prune the behavior graph by eliminating from the model possible attacks that may have occured during the learning phase. Therefore, and since both the behavior graph and the attack graph are not semantically equivalent, we proceed with an intermediary step where we automatically convert the attack graph into an intermediate event graph. Finally, we prune the behavior model from possible attacks by applying standard graph matching and subgraph isomorphism to both behavior graph and the event graph.

To summarize, this paper makes the following contributions.

- We present a novel approach to characterize elementary user interactions with the service. Our approach applies unsupervised clustering in order to build class behavioral graphs, where each graph captures a set of nominal user interactions with the service.
- We bridge the gap between behavioral graphs and attack graphs by introducing an intermediary model, event graphs. It allows to prune the behavioral model from attacks that may have occurred during the learning phase.
- We demonstrate the use of our system through a practical use case, including real world attack scenarios that we were able to qualify using our approach.

This paper is organized as follows. Section 2 summarizes related work on attack detection in web applications. Section 3 provides a review of the main techniques used in the paper. Section 4 provides an overview of our approach. Section 5 presents the technical details, including the main components that constitute our system. Section 6 describes a practical use case. Finally, we conclude in Section 7.

2. RELATED WORK

Related work includes multiple contributions that aim at detecting attacks on web applications [1, 4, 7, 14, 15]. We classify these contributions into two categories. The first category requires access to the source code, i.e. white-box approach, in order to detect and qualify unintended security flaws. It mostly applies to well-known attack classes such as Cross-Site Scripting (XSS) [16], Cross-Site Request Forgery (CSRF) [17] and SQL injection [18]. The second category processes the input and output parameters of an application, assuming little information about its internal structure and configuration, i.e. black-box approach. In the following, we discuss the main advantages and limitations of both techniques.

The white-box approach applies either static or dynamic analysis directly to the source code of an application [32], which requires the source code to be accessible for auditing. MiMoSA [20], Waler [21], and Swaddler [22] are examples of white-box approaches. In particular, MiMoSA and Waler use a similar concept that consists of directly inferring a model from the source code. In order to detect violations of state invariants, MiMoSA and Waler use a model checker. Swaddler, however, analyzes the internal states of an application and learns the relationships between its critical execution points and internal states. It qualifies as an attack all attempts to violate the workflow of the application, which may drive the application into an anomalous state. In addition to static analysis [19], dynamic analysis offers to explore the execution space of an application, using symbolic execution, in order to find unintended vulnerabilities [28, 36]. In general, the white-box approach has multiple limitations. First of all, it requires direct access to the source code, and so it mostly targets specific programming languages such as javascript and PHP. Therefore, target applications are limited to only those that are compatible with the supported programming language. Moreover, the white-box approach suffers from a high false negatives rate, as it mostly detects known and well-qualified attacks and vulnerabilities.

On the other hand, the black-box approach offers an alternative solution that aims at detecting attacks through analyzing the application's output based on a pre-defined set of input variables [23, 7, 12, 24]. In this category, the application is considered as a sealed machine with unobservable internals. Authors in [7] present a state-aware input fuzzer to detect XSS and SQL injection vulnerabilities.

Another tool, BLOCK [12], learns invariants when observing a sequence of HTTP conversations in order to detect authentication bypass attacks. Moreover, authors in [4] model the workflow of an application using a set of HTTP conversations, and then automatically generate and execute test cases using a number of specific attack patterns. The closest to our contribution is AUTHSCAN [14], which infers a model through combining both white-box and black-box techniques. It automatically recovers authentication protocol specifications from web implementations, and so it only detects security flaws that are specific to authentication protocols.

As opposed to current state of the art, we propose an approach that efficiently combines both black-box and white-box techniques. Compared to the recent works, our approach is generic enough so it can detect all security flaws as long as they affect the workflow or behavior of a target application.

3. BACKGROUND

In this section, we briefly recall machine learning, data mining, and graph concepts that we use in our paper. For the sake of clarity, we also point the reader to appropriate references for a complete introduction on those topics.

3.1 Co-clustering

Co-clustering is a clustering technique that leverages the duality between the clustering of both items and features in a given dataset [25]. The co-clustering is an unsupervised learning technique that simultaneously segment the rows and columns of a matrix. A typical example of co-clustering considers the context of document clustering. Hence, co-clustering is associated to finding minimum cut vertex partitions in a bipartite graph between documents and words [25]. Given different document clusters D_1, ..., D_k, then we determine the corresponding word clusters W_1,..., W_k as follows. A given word w_i belongs to the word cluster W_m if its association with the document cluster D_m is greater than its association with any other document cluster. A measure of the association of this word with a document cluster consists on summing the occurences of this word in each document of the cluster.
$W_m = \{w_i : \sum_{j \in D_m} A_{ij} \geq \sum_{j \in D_l} A_{ij}\}$, $\forall\ l = 1, ..., k$ where A_{ij} is the number of occurences of word w_i in document d_j. Hence, each of the word clusters is determined by the document clustering. Also, for a given word clusters W_1,..., W_k, then the document clustering is as follows:
$D_m = \{d_j : \sum_{i \in W_m} A_{ij} \geq \sum_{i \in W_l} A_{ij}\}$, $\forall\ l = 1, ..., k$.

3.2 N-Grams

The n-grams a technique was initially proposed for natural language processing [26], then it was later used in many detection systems [33, 27]. In order to describe data in terms of n-grams, each data object x is represented as a string of symbols from an alphabet A, where A is defined as bytes or tokens. We can extract all substrings of length n that are called the n-grams, by moving a window of n symbols over each object x. These n-grams give rise to a map of a high-dimensional vector space. Each dimension is associated with the occurence of one n-gram. This map Φ can be constructed using the set S of all possible n-grams as follows:

Φ: $x \to \Phi_s(x)$ with $\Phi_s(x)$=$occ(s, x)$ where the function $occ(s, x)$ returns the frequency of the n-gram s in the data object x.

3.3 SubGraph Isomorphism

A matching process between two graphs G_1=(N_1,E_1) and G_2=(N_2,E_2) consists in the determination of a mapping M. This mapping associates nodes of G_1 with nodes of G_2 and vice versa, according to some predefined constraints. Generally, the mapping M is considered as the set of pairs (n, m) with ($n \in G_1$ and $m \in G_2$) that represent the mapping of a node n of G_1 with a node m of G_2. A mapping $M \subset N_1 \times N_2$ is called an isomorphism if M is a bijective function that preserves the structure of the two graphs. A mapping $M \subset N_1 \times N_2$ is called a graph-subgraph isomorphism if M is an isomorphism between G_2 and a subgraph of G_1.

The main challenge is to choose low complexity subgraph isomorphism algorithms suited for matching large graphs, in terms of computation time, and memory storage. Some of the proposed algorithms reduce the overall computational complexity of the matching process by imposing restrictions on the graphs (e.g., polynomial algorithms for trees, planar graphs, or bounded valence graphs) [34]. Other approaches as VF2 algorithm [30] includes a deterministic matching method that verifies both isomorphism and subgraph isomorphism. These later approaches demonstrate a low computation time compared to algorithms in [34, 37].

4. SYSTEM OVERVIEW

The overall architecture of our system is represented in Figure 1. It includes three main modules: the Behavior Graph Generator (BGG), the Attack Graph Mediator (AGM), and the Behavior Graph Pruning (BGP). This section briefly describes the main functionalities provided by each module, while further implementation details are presented in section 5.

4.1 Behavior Graph Generator

The BGG module processes a set of real world user interactions with the service. These interactions may be collected either as raw user http requests, or as access logs produced by the service such as for example in the syslog format. The BGG module applies unsupervised clustering in order to infer different behavior classes, where each class characterizes a common user behavior when connecting to the service. In particular, our system leverages the fact that during a connection to the service, users may perform multiple operations (e.g., http queries), and interact with specific components of the service (e.g., web applications) in specific sequences that determine the user journey through the service.

The main idea is that even though benign users may have different profiles as they interact differently with the service, or that they access different applications implemented by the service, they still share common subsequences of interactions that are related to the business model and the access control restrictions imposed by the service. The BGG module applies unsupervised co-clustering in order to automatically learn these common subsequences, and associate them with observable features that are further represented into a statistical model that captures all user behaviors when they connect to the service. The output of this step is a set of different user behaviors.

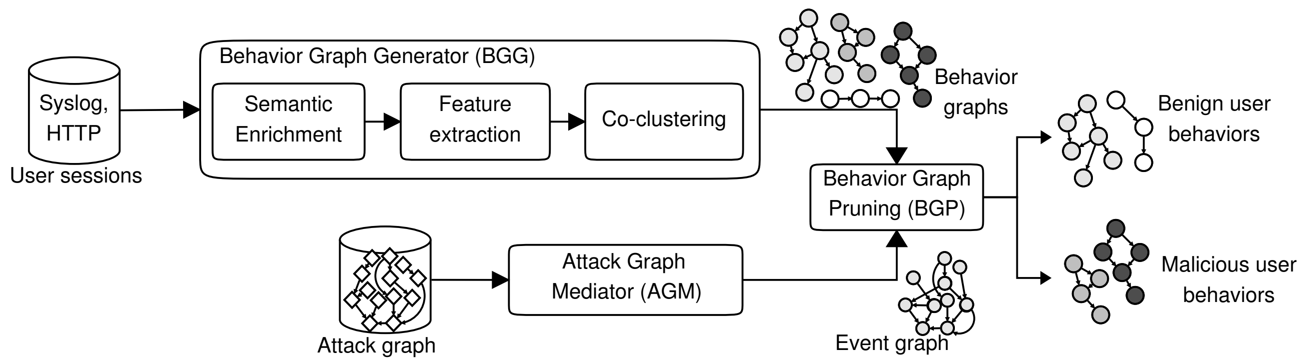


Figure 1: Overview of our approach

Each behavior may be either benign or malicious, depending on the input dataset and its associated ground truth.

Example.
An Internet shopping service may include multiple web applications such as authentication, store, command and payment. Users who connect to the service follow a customer journey that begins with authentication, and further ends at any other stage of the workflow. In particular, users who purchase items on the web site first authenticate to the service, and then access the online store to command their items, before they are redirected to the payment. The authentication, command, and payment applications are mandatory, and so they should appear in all sessions where users purchase items on the web site. However, users may still have different behaviors as they connect to the service, such as spending more time to navigate through the items, adding and retrieving selected items before command, yet also to cancel the operation and leave. To characterize such benign behaviors and separate them from other malicious behaviors such as attacks and fraud, the BGG module processes a set of user interactions with the service. It decomposes each user journey into a set of elementary user requests of length n, and then group sessions together into behavior classes based on the occurrence of common elementary user requests.

4.2 Attack Graph Mediator

In order to separate benign and malicious behaviors, our system leverages a semi-supervised learning approach, using ground truth attack scenarios represented within an attack graph [35]. The rationale for our approach is that it is very difficult, yet practically impossible, to have a representative record of attacks in order to directly apply supervised learning at the BGG module. However, using an attack graph to prune the user behavioral model from possibly malicious behaviors is not straightforward, since attack graphs and behavior graphs are not semantically equivalent. On the one hand, attack graphs represent sequences of flaws and attack procedures that may be executed by an attacker. On the second hand, behavior graphs represent user interactions with the service, and that may be captured either through HTTP requests or raw access logs produced by the service. Hence, the AGM module operates a mediation step where the attack graph is used as input in order to generate an intermediate event graph.

The AGM module iterates over all scenarios in the attack graph. It associates each node in the attack graph with a sequence of probabilistic events that may characterize the occurrence of such attack on the service. The set of all elementary events constitute the event graph, which is further used by the BGP module in order to prune the behavior classes from all malicious behaviors that may have occurred during the learning phase.

4.3 Behavior Graph Pruning

Finally, the BGP module applies sub-graph isomorphism to both the event graph and the user behavior graph. The purpose of this module is twofold. First, it identifies and isolates malicious behaviors and attacks that may have occurred during the learning phase, and that are similar to known attack scenarios represented in the event graph. Second, it identifies all benign user interactions with the service, and associates them with different behavior classes. All further connections to the service that do not fit with any of these behavior classes are identified by the system as new attack behaviors.

5. SYSTEM DESCRIPTION

In this paper, we represent the behavior of a service through a sequence of URL queries that are triggered by a user during her session with the service. Therefore, we consider that all applications and resources supplied by a service are indeed accessed through specific and unique URL instances. Note that we refer to URLs in this paper as a way to represent user behaviors, but our approach may leverage any other type of information that may be found in the logs produced by the service, and that may fully qualify each user interaction with the service.

In this section, we describe the technical details for each of the three components that constitute our system. We also present the data structures, algorithms and procedures along with the appropriate implementation choices. We further describe in Section 6 a practical use case that illustrates the use of our scenario.

5.1 Behavior Graph Generator

In the following, we first provide a formal definition of the class behavior graph and its underlying semantics. We then describe more in detail the implementation of the BGG module.

5.1.1 Behavior Class Definition

A behavior class represents an ordered set of elementary user interactions with a service. It is represented as a directed graph $G= \{N, E\}$, where N is the set of nodes, and E is the set of directed edges. The set N of graph nodes includes all couples (a_i, r_j), where $A= (a_i)$ is a set of all elementary applications supplied by the service and $R= (r_j)$ is a set of all possible URL queries. An edge $e_{1,2}$ specifies that users are redirected from an application a_{i1} where they have performed an URL query r_{j1}, to an application a_{i2} where they can perform another URL query r_{j2}.

5.1.2 Implementation of BGG module

To generate class behaviors, the BGG module implements a three-steps process including first a semantic enrichment of URLs, then a feature extraction through statistic analysis, and finally co-clustering. These steps are described as follows.

Step 1: Semantic Enrichment.

This method consists of enriching specific values in URL queries by generic ones. According to RFC 2616, two URLs are exactly similar if they match through a case-sensitive octet-by-octet comparison. Using the same comparison in our system may lead to an overfitted model. In fact, URL parameters may include user-specific attributes, that will make each URL unique in the context of a given user or session. Therefore, in order to handle this limitation, the BGG module associates each URL instance with an elementary URL token that may match with other semantically equivalent URLs in other user sessions with the service. The semantic enrichment process, that transforms each URL query into a generic URL token, takes into account the following guidelines:

- URL components are handled based on their semantical meaning, so that paths may be compared with paths, domains with domains, and parameters are compared based on their keys and the semantic of their values.

- Domain names and paths point to specific resources or web applications that are supplied by the service. They are handled as string components and matched on an octet-by-octet basis when comparing two URL queries.

- In the URL query part, parameters may indicate service-specific attributes that enable a service to redirect users towards specific web applications. They may also indicate user-specific attributes that enable a service to tune the content delivered to the user, but may not determine the user journey through the service.

Service-specific parameters in two separate URLs are equivalent when they share the same key and value; however two user-specific parameters are equivalent when they share the same key and semantic. For example, in the following we consider URL_1:

https://perso.domain.fr/maf.php?contact=123456&
page=svc-mobile

The "contact" attribute designates the user Id, while the "page" attribute designates the application that is being accessed by the user.

The semantic enrichment process, which replaces specific parameters with generic ones, provides the following URL token T_a:

https://perso.domain.fr/maf.php?contact=[userId]&
page=svc-mobile

Step 2: Feature Extraction.

In the second step, the BGG module extracts features through the use of sequence of tokenized URL queries extracted from each observed session. We choose the n gram analysis technique in order to decompose the list of requests associated with a user connection into an n-gram histogram. The n-gram histogram summarizes the occurence frequency for all elementary user interactions during the connection to the service. n-grams analysis includes the set of techniques and tools that aim at building and processing contiguous sequences of items or occurences of a given process such as test, speech, or random variables [29]. It consists of decomposing a sequence of outcomes for a given process into multiple contiguous subsequences. Compared to the traditional applications of n-gram analysis, we applied this technique in a novel way for a sequence of user requests during a connection to the service. Each specific request designates an access to a specific application or resource supplied by the service. The value of n is an input to the system that is empirically specified, through combining into an elementary sequence every n consecutive user interactions within a same session. Since a given n-gram may appear multiple times during a session, the BGG module assigns occurence frequencies based on the number of times each n-gram sequence appears within the session under consideration. Our system builds a distribution vector v_i of the session i, that represents the occurence frequencies of each n-gram sequence. Later, our system uses the computed n-grams and their distribution over the sessions observed in the learning phase to build an n-gram distribution matrix M_D. Rows in M_D represent the sessions observed during the learning phase, and columns represent all encountered n-grams and their distribution over each session. The resulting matrix is a sparse and high-dimensionality matrix.

Let us denote by g the set of all n-gram sequences; s is the total number of observed sessions; and X_j is the j^{th} segment or j^{th} n-gram, $j \in [1,|g|]$. The distribution vector of session i is v_i=$< f'(X_j, i) >_{j=1}^{j=|g|}$, where f' represents the occurence frequency of an n-gram within a session i. Then the distribution matrix is presented as follows M_D=$[v_i]_{i=1}^{i=s}$.

- Example: Let us assume a session including the following sequence of tokenized URL queries:
$< T_a, T_b, T_c, T_a, T_a, T_b, T_d >$, and n=2. In this case, we have the following decomposition
$< T_aT_b$,T_bT_c,T_cT_a,T_aT_a,T_aT_b, $T_bT_d >$. As a result Table 1 presents the 2-gram distribution frequencies. For the sake of clarity, we refer to the *other tokens* column as the remaining sequence of 2-grams (T_aT_c, T_aT_d, T_bT_a, T_bT_b, T_cT_b, T_cT_c, T_cT_d, T_dT_a, T_dT_b, T_dT_c, T_dT_d). The frequency of these 2-grams in this session is 0.

$2-gram$	T_aT_b	T_bT_c	T_cT_a	T_aT_a	T_aT_b	T_bT_d	*other tokens*
Freq.	2	1	1	1	1	1	0

Table 1: Example of 2-gram distribution frequencies

Step 3: Co-clustering.

In this paper, we draw a dual analogy in order to cluster sessions based upon their n-grams distribution, while also clustering n-grams based on their co-occurence into different sessions.

The output of the second step (feature extraction) is an n-grams distribution matrix, that is used in this co-clustering step in order to build behavior classes. This technique builds m distinct behavioral clusters (classes) $\{C_z\}_{z=1}^{z=m}$, where the number of clusters m is an output of the process.

5.2 Attack Graph Mediator

The second component of our system is an attack graph mediator (AGM) that takes as input an attack graph and generates as output an event graph. We use the same definition of attack graph as the one in [35]. The nodes in the behavior graph represent events that may be characterized through URL requests. Hence, we need to associate attacks/exploits to events. Our model considers the full scope of all known attack paths, and it represents events that are generated by the vulnerability exploits in the attack graph using a directed graph EG=$\{N', E'\}$, where N' is the set of nodes representing the events, and E' is the set of logical transitions between events. We capture relationships among attack graph elements in concise distance measurements, and for this we get inspired by the work done in [35]. When following a certain attack path in the attack graph, the exploits may or not generate events that are tracked in the URLs. In fact, some attacks could not be tracked in the service logs since an attacker does not intereact directly with the service such as for example interacting directly with the victim user (e.g. phishing attack). However, when an attacker interacts directly with the service, in this case he may be tracked in the logs of that service.

Algorithm 1 details the main process of the AGM component. It takes as input an attack graph AG, and generates an event graph EG. First, we consider a node x, and y is a child node of x in the attack graph AG. Node x triggers k events ($f(x)$= $\{v1\}_{i=0}^{i=k}$) where f is a function associating to each node in the attack graph a list of events. Node y triggers l events such that ($f(y)$= $\{v'\}_{j=0}^{j=l}$) (line 3). In this case, Algorithm 1 updates the event graph by adding the list of nodes (line 5), and by adding also events triggered by x and y (line 6). If the node x in the attack graph does not trigger any event (line 7), in this case we search in the attack Graph AG the last parent node triggering events ($T \leftarrow search\_for\_last\_parents\_nodes\_triggering\_events$ (AG, x)) (line 10). T represents the list of events triggered by the parent node of x. For the updated event graph, Algorithm 1 associates to each event triggered by node y the list of events in T ($\mathcal{E}' \leftarrow \mathcal{E}'+ \sum_m \sum_j e(T_m, v'_j)$) (line 11). By exploring all the nodes of the attack graph, the AGM component updates the event graph EG. Then, we apply the same process to the next child node of x.

Example.

For the sake of clarity, we present the example illustrated in Figure 2. This example presents an attack graph with five nodes $Ex1$, $Ex2$, $Ex3$, $Ex4$, and $Ex5$. Each node represents an exploit/vulnerability. We suppose that an initial input event exists as $E1$, and which is associated with the Exploit $Ex1$. Events $E2$, and $E3$ are both considered to be associated with Exploit $Ex2$.

Algorithm 1: Attack Graph Mediator

```
1  Input: Attack Graph $AG$= { $\mathcal{N}$, $\mathcal{E}$ };
   Output: Event Graph $EG$={ $\mathcal{N}'$, $\mathcal{E}'$ };
2  Initialization;
   $EG$=∅ ;
   $\mathcal{N}'$=0 ;
   $\mathcal{E}'$=∅ ;
   Let $f$: injective function, $e$: function associating two nodes in a graph ;
   $x \in \mathcal{N}$ ;
   $f(x)$= $\{v1\}_{i=0}^{i=k}$ //Node $x$ triggers $k$ events ;
   while $y$ is $Child\_node(AG, x)$ do
3  $f(y)$= $\{v'\}_{j=0}^{j=l}$ //Node $y$ triggers $l$ events ;
4  eIf$f(x) \neq \varnothing$ && $f(y) \neq \varnothing$
5  $|\mathcal{N}'| \leftarrow |\mathcal{N}'| + |f(x)| + |f(y)|$ ;
6  $\mathcal{E}' \leftarrow \mathcal{E}' + \sum_i \sum_j e(v1_i, v'_j)$ ;
7  if $(f(x) = \varnothing)$ then
8      $|\mathcal{N}'| \leftarrow |\mathcal{N}'| + |f(y)|$ ;
9      $T \leftarrow$ $search\_for\_last\_parents\_nodes\_triggering\_events$ $(AG, x)$;
10     $\mathcal{E}' \leftarrow \mathcal{E}' + \sum_m \sum_j e(T_m, v'_j)$ //$m$ index of $T$ ;
11 $visit\_next\_child\_node(G, x, y)$ ;
```

By applying Algorithm 1, we add the nodes $E1$, $E2$, and $E3$ to the event graph, along with the appropriate set of Edges = { $E1 \rightarrow E2$, $E1 \rightarrow E3$ }. Event $E4$ is added as a child to the events $E2$ and $E3$, since $E4$ corresponds to the Exploit $Ex3$. We update the list of edges by adding { $E2 \rightarrow E4$, $E3 \rightarrow E4$ }. Then nodes $E5$ and $E6$ correspond respectively to $Ex4$ and $Ex5$. Hence, one event path is added between $E4$ and $E5$, and another event path between $E4$ and $E6$.

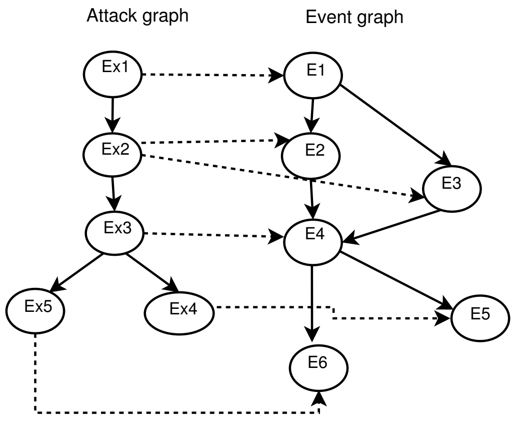


Figure 2: Example of AGM component execution

5.3 Behavior Graph Pruning

The Behavior Graph Pruning (BGP) component leverages subgraph isomorphism techniques. We match all the class behavior graphs against the event graph. Whenever a match is found, the corresponding class behavior is detected as an attack, and we can extract this attack from the behavior graph. We introduce Algorithm 2 that corresponds to the graph pruning. The objective is to detect whether a behavior class is malicious or not.

To this end, Algorithm 2 applies the subgraph isomorphism test (line 2). This test takes two graphs C_i and EG as input, and then determines whether EG contains a subgraph that is isomorphic to C_i.

In particular, when the test fails, we consider the nodes of the class behavior C_i as not being part of a malicious events sequence, and therefore C_i captures a benign user behavior. Otherwise we consider the class C_i to be malicious because it is isomorphic, in terms of subgraph isomorphism, to the malicious event graph EG.

Algorithm 2: Behavior Graph Pruning

```
1 Input: $C_i$: behavior Class Graph, $EG$: Event Graph ;
2 $map \leftarrow subgraph\_isomorphism(C_i$, EG) ;
3 if map =NULL return false ;
4 else return true ;
```

6. EXPERIMENTS

This section describes the design of our experiments, as well as the dataset that we used in order to build and validate our approach. First, we build class behaviors using a set of collected logs. Then, we generate an attack graph from risk analysis results. We assess the feasibility of our greybox approach by pruning the behavior classes from attacks. Then, we show how our approach detects new undiscovered attacks. Last of all we draw some conclusions and point out ways in which this work can be further extended.

6.1 Use Case Scenario

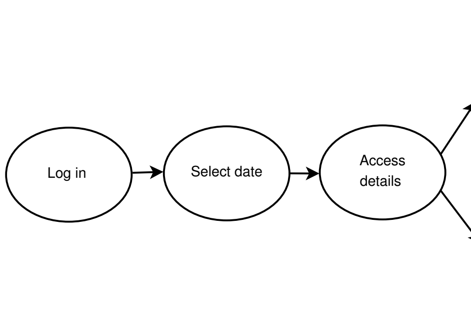
We demonstrate the use of our system through the example of an online service. To do so we have collaborated with a service provider. In the black-box based approach, the analysts provided us with a fully anonymized service logs that we used as input to generate the class behaviors. The dataset was collected over a period of six weeks, including 33.7 GB of data in the syslog format, associated with more than 1.3 million active user sessions. The logs are created by an online service portal providing access to different applications, including email, video on demand, online games, digital shop, weather forecast, and news feeds. The format of these logs is illustrated as follows. Each log instance includes a timestamp, anonymized user identity and remote IP address, and the landing URL of the associated service. Note that the fact of these logs being fully and irreversibly anonymized does not limit the usage of our approach, since our system does not need to process the exact user identities at any time. In the white-box approach, the analysts provided us with their internal risk analysis. They also pointed out in the logs of the service certain security flaws that they were able to detect and qualify. For each identified security risk, they explain to us the process and how to identify such attacks. The list of security risks includes the video online gaming purchase, the cookie hijack, the cross-site forgery attack, and the identity theft. Hence from these obtained attacks, our system generates different attack graph paths, and then it builds the corresponding event graphs. In the following, for the sake of clarity, we detail some of the identified attacks. Our aim is not to mention all the encountered security attacks; however we shed a light on the way our approach detects these attacks and discover new ones.

One contribution of the paper resides in the capacity of our system to characterize all requests during a given session as a logical sequence that characterizes the user journey through the service. For instance, from our dataset, each user request includes an URL associated with a specific application that is supplied by the service. It indicates the relative path to the application and the parameters that make possible for users to access specific assets provided by this application. After collecting the set of logs, the BGG module generates the different class behaviors from these logs. First, the system applies a semantic enrichment. Then, the BGG component proceeds with a feature selection technique using the n-grams analysis. Later, our system builds n grams sequences of length n through iteratively combining every n consecutive user requests that appear within the same session.

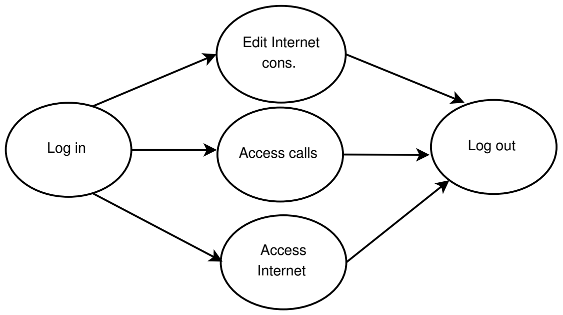
Since a given n-gram may appear multiple times during a session, our system assigns occurence frequencies based on the number of times each n-gram sequence occurs within the session under consideration. The resulting n grams are further used by the system in order to generate multiple behavioral clusters. Each cluster captures user behaviors when connecting to the service. Note that for small values of the parameter n, the system will be able to capture only short user interaction sequences with the service. For instance, in case where n is set to 1, the system will be able to capture abnormal behaviors such as brute force attacks where an attacker triggers a large number of queries that relate to a single or small set of tokens. It may also capture specific attacks where an attacker injects malicious code in the URL, in such a way it would match with no other token that is built in the black-box approach. However, it may not capture other low level attacks such as cross-site request forgery, where unauthorized requests are transmitted from a victim user and that may require a more elaborated representation of interaction sequences with the service. On the other hand, high values for the parameter n may also lead to an overfitted model where only few sessions would share common sequences. While such high values considerably reduce false negatives, they may also contribute to a higher false positives rate since every small, but normal deviation during runtime will be considered as abnormal by the system. We carried out a set of scenarios in order to choose the optimal value of n inducing a low false positive rate and a low false negative rate. In our case, we evaluated different values of n varying from 1 to 5, and we empirically set the value of n to 2.

After generating the set of ordered sequences of elementary user interactions, the BGG module applies the co-clustering algorithm to generate clusters of user behaviors. We may refer to the AnyDBC algorithm [31] that achieves computation time less than $\theta(o^2)$ and memory storage in the order of $\theta(vo)$, where o the number of objects and v is the number of initial nodes of the cluster graph in AnyDBC algorithm. Our system is able to characterize a list of 32 class behaviors corresponding to a log file of 33.7 GB. We discuss in the following five class behaviors $Class$ #1, $Class$ #2, $Class$ #3, $Class$ #4, and $Class$ #5.

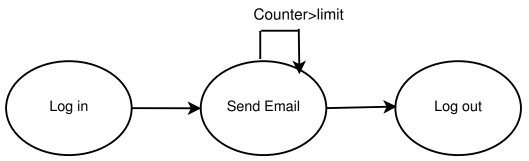
Figure 3 illustrates the graphical representation of these class behaviors.



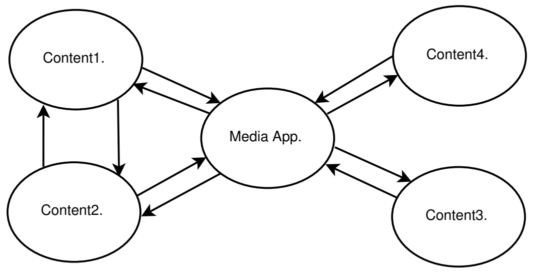
(a) Class #1: Billing Service



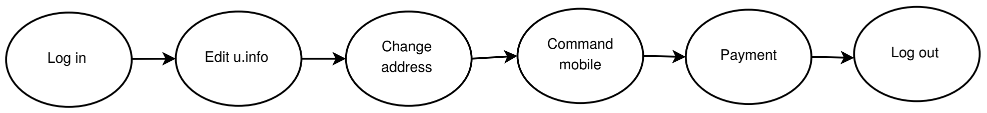
(b) Class #2: Access to Calls and Internet Consumption



(c) Class #3: Access to E-Mail Service



(d) Class #4: Access to Media Service



(e) Class #5: Mobile Phone Change

Figure 3: Class Behaviors

- *Class #1*: This class behavior represents user interactions when connecting to the billing service. The graph consists of five nodes. The first node represents the connection to the service ("*Log in*"). Then users are redirected to select a given period ("Select date"). After that, users either access to their bill ("*Access details*") or they could download the bill "*Download*". After the two previous interactions, users log out ("*Log out*").

- *Class #2*: This class behavior captures the workflow for the application that provides access to users' calling history and their internet consumption. The graph includes five nodes. First, users connect to the service ("*Log in*"). Then, users have the following options: i) they can edit their internet consumption ("*Edit. Internet cons.*") in order for example to switch to another Internet offer, ii) they access to their calls consumption ("*Access calls*"), or iii) they access to their Internet consumption over a selected period of time ("*Access Internet*").

- *Class #3*: This class behavior represents the way users access to the email service, including their ability to send any number of emails (counter). Interrestingly in our dataset, we observed a statistically representative number of user sessions where the number of sent emails was constantly above a given threshold ($counter > limit$).

- *Class #4*: This class behavior depicts the way users access to the media service, that allows them to pay when accessing contents. As presented in Figure 6.1, this service enables them to switch from a media content to another by passing through one specific component ("*Media App.*").

- *Class #5*: This class behavior represents the mobile phone change. The graph consists of six nodes. After connecting to the service, users modify the delivery address in the profile account ("*Change address*"). After that, users change their mobile phones ("*Command mobile*"). To do so, they command the phone and then they are redirected to the payment service. Finally, the customers have the payment confirmation status for the new mobile phone ("*Payment*").

On the other hand, the AGM module takes as input an attack graph (Section 4.2), that was built using the results of risk analysis that was conducted by the service provider. For the sake of clarity, we mention here three attacks (we refer the reader to Figure 4).

- Attack #1: We refer to this attack as a Video Game Purchase Fraud. The attack path consists of seven nodes. To achieve his goal, an attacker proceeds first by executing a cookie hijack. Then, the attacker impersonates the identity of another victim user. After connecting to the service, the attacker is able to edit the user info ("*Modify Info*"), and then the attacker changes his email address ("*Set Email Address*").

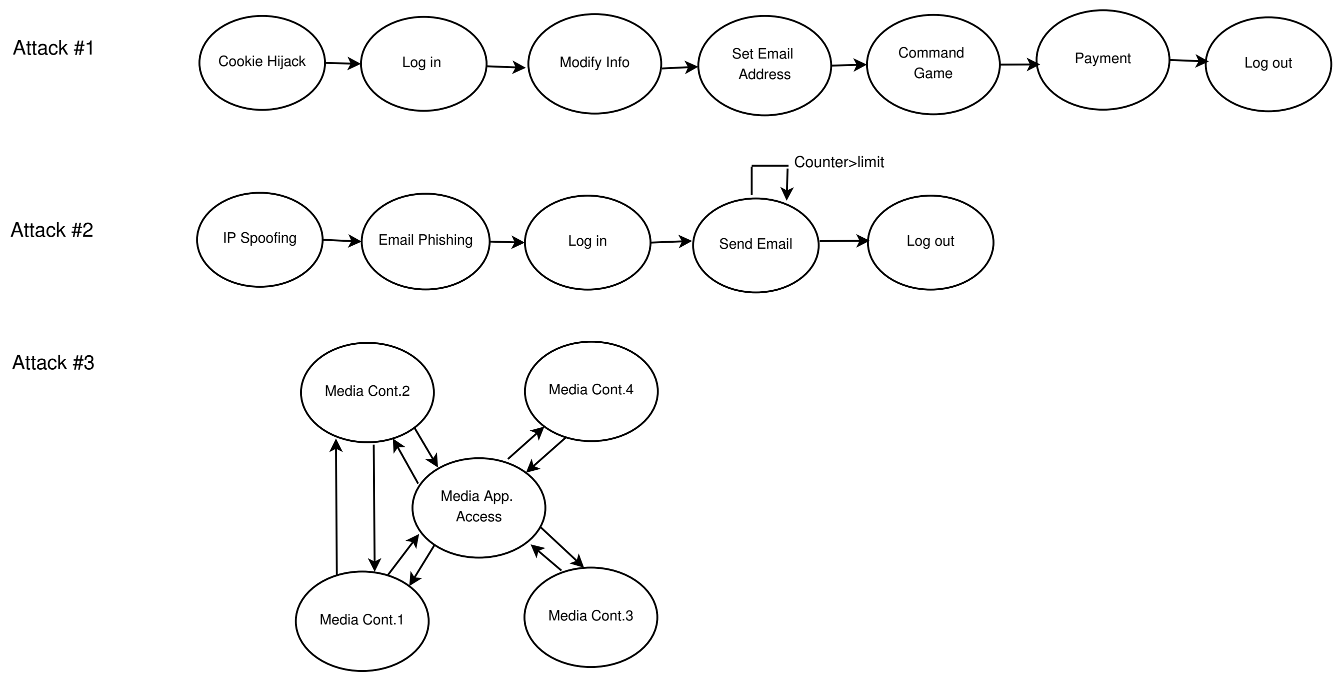


Figure 4: Three Attack Paths

Later, the attacker commands a game, and receives the confirmation payment. By exploiting the vulnerability of the service, the attacker will let the victim user be charged for the game, and delivered to the new email address.

- Attack #2: The attacker connects to the WiFi of the victim user, and behaves as the legitimate user ("*IP spoofing*"). Then he proceeds with an email phishing exploit. In this scenario, the attacker connects to the service, may access all the account details for the victim user, and may then send as many phishing emails as needed on behalf of the victim user account.

- Attack #3: The attacker connects to the media service, and he exploits a vulnerability in the application that allows him to access from a media content ("*Media Cont.*1") to another ("*Media Cont.*2") without access to the billing service ("*Media.App.Access*").

After detailing the workflow of the different malicious behaviors, the AGM component refers to Algorithm 1 that takes as input the three different attack paths, and generates the corresponding event graphs as described in Section 5.2. To do so, for each exploit in the attack graph, the AGM module builds the event graphs. We consider here the event graphs corresponding to Attack #1, Attack #2 and Attack #3 presented respectively in Figure 5, Figure 6, and Figure 7. The dotted lines in all figures represent the mapping function associating nodes to events (function f in Algorithm 1).

- Event Graph of Attack #1: When an attacker executes the cookie hijack exploit, the AGM module is not able to find it in the logs (since there is no direct interaction with the service).

  For the second exploit "*Log in*" corresponds an event "*Log in*" in the event graph. When the attacker modifies user info, this action is detected in the logs, and generates an event "*Edit u.info*" in service logs. Finally, for each action "*Command Game*", "*Payment*", and "*Log out*", it triggers respectively events that are observable in the logs "*Command Game*", "*Payment*", and "*Log out*". As output to Algorithm 1, the AGM module generates the event graph corresponding to the Attack #1 as presented in Figure 5.

- Event Graph of Attack #2: An attacker spoofs the identity of a legitimate customer and may then connects to the email service sending phishing emails. The number of phishing emails in this attack is expected to exceed a predefined threshold value over a given period of time, and which is empirically specified by the analysts (Figure 6).

- Event Graph of Attack #3: This graph corresponds to an attacker that switches from a media content to another without accessing the billing service. In the logs file, the AGM module is able to associate the actions of an attacker to events found in the logs (we refer the reader to Figure 7). For instance, when an attacker connects to the service, he generates events in the service logs. In our example, for each interaction done by the attacker, it corresponds to an event in the service logs. For instance for "*Media App.Access*" exploit in the attack graph, it corresponds an event in the logs that is "*Media App*". In the same manner, Algorithm 1 associates for each action that allows the access to a content an event (for example "*Content*1." in the event graph). At the end, the AGM module builds the event graph corresponding to the third attack.

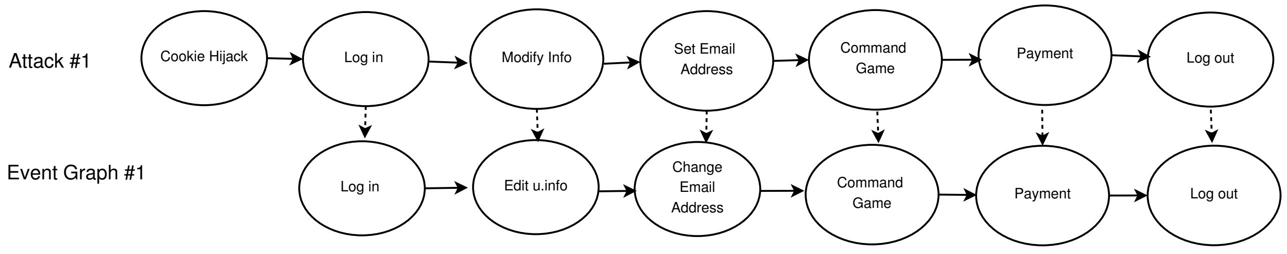


Figure 5: Attack #1 and Event Graph #1

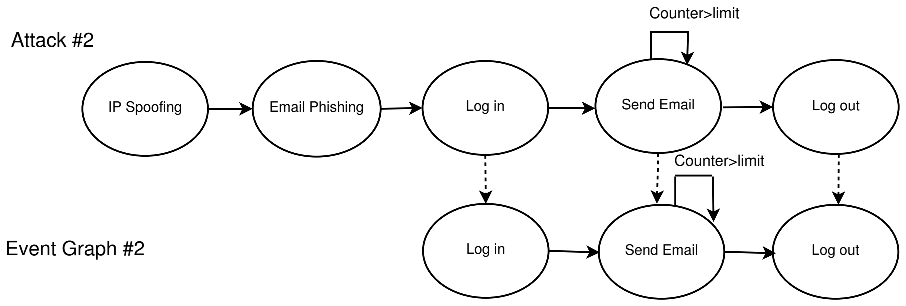


Figure 6: Attack #2 and Event Graph #2

After generating the different class behaviors and the event graphs, the BGP component applies Algorithm 2 (Section 5) between each class behavior and the event graphs. We choose the VF2 algorithm as a deterministic matching method for verifying subgraph isomorphism [30]. The algorithm has general validity since no constraints are imposed on graph topology. The memory requirement with respect to the number of nodes N in the graph is $\theta(N)$ and the time complexity is $\theta(N^2)$. The size of our class behaviors varying from 2 to 13 nodes are sufficient enough to execute the subgraph isomorphism test.

Let us first consider the $Class\ \#4$ that represents user interactions when connecting to the media service. In fact, node "*Media App*" in the Event Graph #3 is equivalent to the node "*Media App*" in $Class\ \#4$. By comparing the different nodes of $Class\ \#4$, Event Graph #3 is isomorphic to $Class\ \#4$. Hence our system is able to detect that $Class\ \#4$ corresponds to an attack previously defined in the attack graph. In the next step, our system applies the subgraph isomorphism test between the Event Graph #2 and the $Class\ \#3$. The output of the algorithm confirms that Event Graph #2 and the $Class\ \#3$ are matching. Thus, from this test our system identifies $Class\ \#3$ as an attack.

Another interesting point to mention is when our system checks the Event Graph #1 and the $Class\ \#5$. Algorithm 2 verifies whether there is an isomorphism between the event graph and the class behavior. As our system applies a semantic enrichment (Section 5.1) and then an attack graph mediation (Section 5.2), the event graphs and the class behavior graphs are semantically equivalent. For instance, the two nodes "*Log in*" in the two graphs are equivalent. Then, "$Edit\ u.\ info$" is the same for the two graphs. In the class behavior graph, the node "*Change address*" is equivalent to the node "*Change Email Address*", since in the semantic enrichment step, the two parameters (address, email address) are user-attributes.

In the same way, the two nodes "$Command\ Mobile$" and "$Command\ Game$" are also semantically equivalent since "*Game*" and "$Mobile$" are service-attributes. As a result, the class behavior corresponding to a video game online purchase $Class\ \#2$ is isomorphic to the attack path of mobile phone purchase. By consequence, $Class\ \#5$ is an attack that was not previously defined in the attack graph; however with the technique of graph isomorphism, the BGP component is able to detect even new attacks. Furthermore, an interesting outcome is that our system is able to update the attack graph with the new discovered attack, and also be able to notify the analysts about this attack.

6.2 Discussion

Our system is able to identify malicious user interactions and even new unknown threats. To do so, our work leverages the use of a black-box and white-box approach in a novel way. The efficiency of our system could be increased by further improving the accuracy of the risk analysis phase. By consequence, our system is able to identify attacks that have the similar behavior as ones defined by the experts, and even new attacks. It is also interesting to notice that our system achieves better performances in terms of attack detection when collecting logs from multiple heterogenous sources (network, servers, applications, etc.), and this may yield a better understanding of how malicious users behave.

In terms of security robustness, let us consider an attacker that executes a mimicry attack in order to evade detection while achieving the attacker's goals. When there is a lack of correlation between logs, the attacker could evade detection in particular when the executed attack steps do not require direct interaction with the service. In other words, the attacker can examine the set of paths that will not trigger any alarms/events, and look for one such cases where the malicious interaction can be inserted in the system.

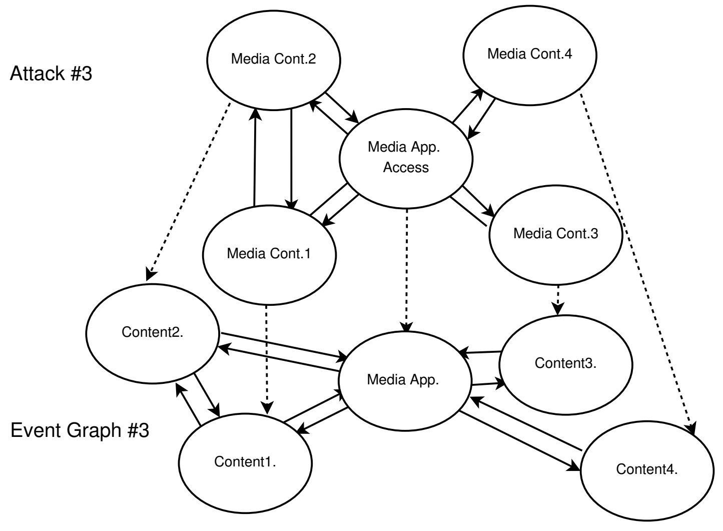


Figure 7: Attack #3 and Event Graph #3

In order to mitigate the impact of this attack, we should have a complete picture of the system by correlating data from different heterogenous sources.

7. CONCLUSION

This paper presents a system that monitors user interactions with online services and also prunes the behavioral graphs from attacks during the learning phase. We do not rely on manual detection of attacks during the learning phase; however we propose a grey-box based approach that combines both advantages of black-box and white-box approach. Using the black-box based approach, we generate different classes of user behaviors using unsupervised machine learning algorithm. In the white-box based approach, we identify some attacks collected from a partial knowledge database in order to generate an attack graph mediator. Then, we proceed with a graph pruning step, using standard graph isomorphism techniques, in order to remove attacks from the behavioral graphs. Finally, we demonstrate the use of our system through a practical use case.

8. REFERENCES

[1] G. Stringhini, P. Mourlanne, G. Jacob, M. Egele, C. Kruegel, and G. Vigna. Evilcohort: Detecting communities of malicious accounts on online services. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 563–578, Washington, D.C., USA, 2015.

[2] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Zigzag: Automatically hardening web applications against client-side validation vulnerabilities. In *24th USENIX Security Symposium*, pages 737–752, Washington, D.C., USA, 2015.

[3] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad. Webwitness: Investigating, categorizing, and mitigating malware download paths. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 1025–1040, Washington, D.C., USA, 2015.

[4] G. Pellgrino and D. Balzarotti. Toward black-box detection of logic flaws in web applications. In *Proceeding of the Network and Distributed System Security Symposium NDSS*, San Diego, USA, 2014.

[5] S. Marchal, K. Saari, N. Singh, and N. Asokan. Know your phish: Novel techniques for detecting phishing sites and their targets. In *Proceeding of the ICDCS*, pages 323-333, Nara-Japan, 2016.

[6] B. Carminati, E. Ferrari, and P. C. Hung. Web service composition: A security perspective. In *International Workshop on Challenges in Web Information Retrieval and Integration*, Tokyo, Japan, pages 248–253, 2005.

[7] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of USENIX Security Symposium (USENIX Security 12)*, pages 523–538, Bellevue, WA, USA, 2012.

[8] D. Canali, D. Balzarotti, and A. Francillon. The role of web hosting providers in detecting compromised websites. In *Proceedings of the World Wide Web Conference WWW*, pages 177-188, Rio de Janeiro, Brazil, 2013

[9] T. Scholte, D. Balzarotti, and E. Kirda. Have things changed now? an empirical study on input validation vulnerabilities in web applications. *Elsevier Computer.Security*, 31(3):344-356, 2012.

[10] D. Fett, R. Küsters, and G. Schmitz. Spresso: A secure, privacy-respecting single sign-on system for the web. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1358-1369, Denver, CO, USA, 2015.

[11] N. Kheir, N. S. Diop, S.-Y. Loui, and V. Frey. Poster abstract: Detecting malicious behaviors through analysis of user interaction sequences. In *Proceedings of the International Conference on Recent Advances in Intrusion Detection RAID*, Kyoto, Japan, pages 1-2, 2015.

[12] X. Li and Y. Xue. Block: A black-box approach for detection of state violation attacks towards web applications. In *Proceedings of the 27th Annual Computer Security Applications Conference ACSAC*, pages 247–256, Orlando, FA, USA, 2011.

[13] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[14] G. Bai, Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. Dong. Authscan: Automatic extraction of web authentication protocols from implementations. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, 2013.

[15] K. Veeramachaneni, I. Arnaldo, A. Cuesta-Infante, V. Korrapati, C. Bassias, and K. Li. Ai2: Training a big data machine to defend. In *IEEE International Conference on Big Data Security*, pages 1–13, New York, NY, USA, 2016.

[16] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, 2007.

[17] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *IEEE Symposium on Security and Privacy (SP)*, pages 332–345, Oakland, CA, USA, 2010.

[18] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Annual International Computer Software and Applications Conference, COMPSAC*, Beijing, China, pages 87–96, 2007.

[19] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. *SIGPLAN Not*, page 2002.

[20] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM Conference on Computer and Communications Security CCS*, pages 25-35, New York, NY, USA, 2007.

[21] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security, pages 10–10, Berkeley, CA, USA, 2010.

[22] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of the International Conference on Recent Advances in Intrusion Detection RAID*, pages 63–86, 2007.

[23] A. Armando, G. Pellegrino, R. Carbone, A. Merlo, and D. Balzarotti. From model-checking to automated testing of security protocols: Bridging the gap. In *6th International Conference on Tests and Proofs*, 2012.

[24] A. Sudhodanan, A. Armando, L. Compagna, and R. Carbone. Attack patterns for black-box security testing of multi-party web applications. In *In Proceeding of the Network and Distributed System Security Symposium NDSS*, San Diego, CA, USA, 2016.

[25] P. Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer, 2006.

[26] J. H. Martin and D. Jurafsky. Speech and language processing. *International Edition*, 710, 2000.

[27] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 41–42, 2004.

[28] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International Conference on Model Checking Software*, pages 2–23, 2005.

[29] W. B. Cavnar, J. M. Trenkle, et al. N-gram-based text categorization. *Ann Arbor MI*, 48113(2):161–175, 1994.

[30] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[31] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 89–98, 2003.

[32] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, pages 151–166, 2008.

[33] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog. Fileprints: Identifying file types by n-gram analysis. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pages 64–71, 2005.

[34] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of computer and system sciences*, 25(1):42–65, 1982.

[35] S. Noel, E. Robertson, and S. Jajodia. Correlating intrusion events and building attack scenarios through attack graph distances. In *Proceedings of the Annual Computer Security Applications Conference ACSAC*, pages 350–359, Tucson, AZ, USA, 2004.

[36] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the IEEE Symposium on Security and Privacy SP*, pages 513–528, Washington, DC, USA, 2010.

[37] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.