Aegis: Automatic Enforcement of Security Policies in Workflow-driven Web Applications[*]

Luca Compagna
SAP Labs France
luca.compagna@sap.com

Daniel R. dos Santos
Fondazione Bruno Kessler
SAP Labs France
University of Trento
dossantos@fbk.eu

Serena Elisa Ponta
SAP Labs France
serena.ponta@sap.com

Silvio Ranise
Fondazione Bruno Kessler
ranise@fbk.eu

ABSTRACT

Organizations often expose business processes and services as web applications. Improper enforcement of security policies in these applications leads to business logic vulnerabilities that are hard to find and may have dramatic security implications. Aegis is a tool to automatically synthesize run-time monitors to enforce control-flow and data-flow integrity, as well as authorization policies and constraints in web applications. The enforcement of these properties can mitigate attacks, e.g., authorization bypass and workflow violations, while allowing regulatory compliance in the form of, e.g., Separation of Duty. Aegis is capable of guaranteeing business continuity while enforcing the security policies. We evaluate Aegis on a set of real-world applications, assessing the enforcement of policies, mitigation of vulnerabilities, and performance overhead.

Keywords

Web Application; Policy Enforcement; Workflow Satisfiability

1. INTRODUCTION

Web applications are one of the preferred ways of exposing business processes and services to users. Many web applications implement workflows, i.e. there is a pre-defined sequence of tasks that must be performed by users to reach a goal [1]. If an application does not correctly enforce its workflows, attackers can exploit this vulnerability to subvert it. In an e-commerce application, for instance, users must *Select products*, *Checkout*, *Enter shipping information*, *Pay*, and *Confirm*. If the application does not verify that user actions follow this sequence, a user can, e.g., skip the payment step and receive products without paying. Control-flow integrity, i.e. the enforcement of an application's workflow, has been used in web applications to prevent workflow attacks and others, e.g., forceful browsing and race conditions [3].

Data-flow integrity is also crucial and incorrect enforcement can lead to vulnerabilities where, e.g., a user can change the price of a product being purchased to pay less for it [20]. This kind of vulnerability is even more prominent in multi-party scenarios, where a user receives data from one party and must relay it to another party. Several vulnerabilities have been discovered in recent years due to improper enforcement of data-flow integrity [20, 12, 14].

Besides control- and data-flow integrity, access control is fundamental whenever users must access only data and functionalities that they are authorized to by a given policy. Access control vulnerabilities are common and hard to find [15]. Moreover, some web applications implement collaborative work, in which users work together to complete a workflow. Examples are Enterprise Resource Planning (ERP) software and e-health applications. In these applications, not only it is important to enforce authorization policies, but it may also be necessary to support authorization constraints, which impose more restrictions on what users can do at run-time. Examples of such constraints are Separation or Binding of Duty (SoD or BoD), requiring two different users (same user, respectively) to execute a pair of tasks. These constraints can be used to avoid errors and frauds in applications that must follow compliance rules. Nonetheless, none of the applications we experimented with provided support for an easy to use, declarative specification of constraints. Without declarative specifications and proper enforcement, authorization constraints have to be implemented as application code embedded into each task or translated to static assignments in the authorization policy. Both solutions are error-prone and can hardly scale.

Even with suitable specification and enforcement mechanisms, support for authorization policies and constraints may lead to situations where an application workflow cannot be completed because no user can execute an action without violating them, thereby hindering business continuity. Determining if such a situation can be avoided, i.e. if a workflow can be completed in the presence of a policy and constraints, is known as the Workflow Satisfiability Problem (WSP) [18]. The WSP has received much attention in the workflow security community, but, to the best of our knowledge, has never

[*]This work has been partly supported by the EU under grant 317387 SECENTIS (FP7-PEOPLE-2012-ITN).

CODASPY'17, March 22 - 24, 2017, Scottsdale, AZ, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4523-1/17/03…$15.00
DOI: http://dx.doi.org/10.1145/3029806.3029813

been considered in web applications. Transferring WSP solutions to the web domain is not trivial, since these solutions rely on workflow models and a workflow management system to handle the control-flow of tasks and to provide an interface for users to request task executions, elements which are frequently not available for web applications.

In this paper, we present Aegis[1], a tool to synthesize run-time monitors for web applications that are capable of automatically (i) enforcing security policies composed of combinations of control- and data-flow integrity, authorization policies and constraints; and (ii) solving the run-time version of the WSP by granting or denying requests of users to perform tasks based on the satisfaction of the policy and constraints and the possibility to terminate the current workflow instance. Aegis is based on [2], where a technique to synthesize run-time monitors that solve the WSP for security-sensitive workflows was presented. We extend [2] by supporting data integrity. To synthesize a monitor, Aegis first infers, using process mining [17], workflow models of the target application from a set of HTTP traces representing user actions. Traces must be manually edited to contain only actions that should be controlled by the monitor. Inferred models are Petri nets [11] labeled with HTTP requests representing tasks and annotated with data-flow properties obtained by using a set of heuristics based on differential analysis (as in, e.g., [20, 14]).

The main contributions of this paper are the description and implementation of Aegis, as well as an empirical evaluation on a set of relevant applications. The rest of this paper is organized as follows: Section 2 presents an overview using motivating examples; Section 3 details the three steps of the technique; Section 4 shows the implementation and evaluation of our work; and Section 5 concludes the paper.

2. OVERVIEW

Aegis synthesizes run-time monitors for workflow-driven web applications, i.e. applications implementing business processes and customer services as workflows. Hereafter, *web application* is used as an abbreviation for *workflow-driven web application*, unless stated otherwise.

A monitor synthesized by Aegis can enforce three security properties: authorization policies ($\mathcal{P}$), defining which users are entitled to perform which tasks; authorization constraints ($\mathcal{C}$), defining run-time restrictions on the execution of tasks, e.g., SoD; and control- and data-flow integrity ($\mathcal{I}$), specifying the authorized control-flow paths that the application must follow, as well as data invariants. Different web applications have different enforcement needs, which requires the synthesis of different configurations of monitors, depending on which properties are switched on or off. We identify each configuration as a tuple containing the active properties, e.g., $\langle\mathcal{P},\mathcal{C},\mathcal{I}\rangle$, $\langle\mathcal{P},\mathcal{I}\rangle$, $\langle\mathcal{C},\mathcal{I}\rangle$, $\langle\mathcal{I}\rangle$. Control- and data-flow integrity are in the same category because it is not realistic that an application needs to enforce one and not the other.

Aegis takes as input sets of HTTP traces representing user actions executed while interacting with a target web application. It synthesizes an external monitor to be used by a proxy sitting between users and the application. Each set of input traces is produced by a user simulating real clients completing a workflow as foreseen by the application ("good traces"). The monitor only enforces those workflows given in input, having no impact on the rest of the application. Traces can be collected using test automation tools such as Selenium[2] or ZAP[3] and must be manually edited to contain only critical tasks. After trace collection, the whole technique is fully automated.

[1]Aegis was the mythological shield carried by Athena.

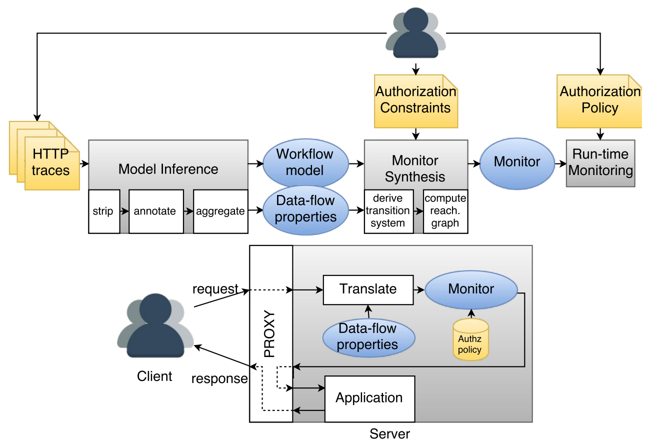

Figure 1: Overview of the technique

Figure 1 shows an overview of Aegis. The top of the Figure shows the entire approach, where rectangles represent the three main steps, yellow notes are inputs, and ovals are generated artifacts. The bottom of the Figure details the internals of the *Run-time Monitoring* component. The three main steps are the following.

1. Model Inference. The set of *HTTP traces* is automatically *stripped* of all information except request and response URLs, headers, and bodies; each request and response is *annotated* with *data-flow properties* inferred by a set of heuristics; traces are *aggregated* into a file called event log; and a process mining tool takes the log as input to generate a Petri net *workflow model* whose transitions are labeled by the annotated requests.

2. Monitor Synthesis. Given a workflow model, the user specifies the *Authorization Constraints* to be enforced (if any) and whether an *Authorization Policy* will be provided at run-time. Control- and data-flow integrity are obtained automatically from the model and are always enforced. The workflow model is presented in a convenient BPMN [19] notation, and the specification of constraints is done graphically. A run-time monitor capable of enforcing the chosen properties is synthesized by translating the model to a symbolic *transition system* (the translation among BPMN, Petri nets, and transition systems is automatic [2]) and computing a *reachability graph* that represents all possible valid executions of the workflow by symbolic users, allowing us to support different authorization policies at run-time. The *Monitor* is a set of queries derived from the graph.

3. Run-time Monitoring. A reverse *proxy* is instantiated with the synthesized monitor and a concrete authorization policy (when provided by the user). It sits between users and the application, filters *requests* and *translates* them to the monitor. The monitor enforces the properties defined in step 2, granting a request if the control-flow is respected, the data-flow invariants hold ($\mathcal{I}$), the user issuing the request is

[2]http://www.seleniumhq.org/
[3]https://goo.gl/XvxKd1

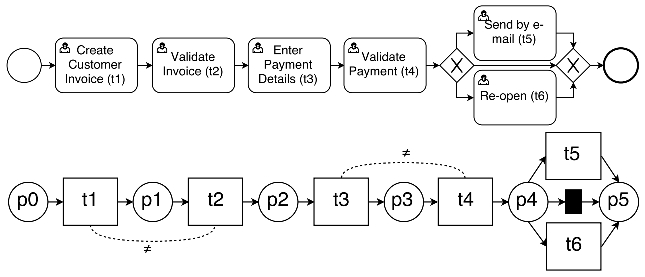

Figure 2: Customer invoice process in BPMN (top) and as a Petri net (bottom)

authorized by the policy ($\mathcal{P}$), the authorization constraints are not violated ($\mathcal{C}$) and the current instance execution can still terminate. The proxy, based on the response from the monitor, may forward requests to the application or drop them to prevent the violation of some property.

A single application may implement several workflows. Steps 1 and 2 are performed for each workflow to be monitored, generating one monitor per workflow. Step 3 uses all the synthesized monitors and queries the correct one depending on the incoming request. Requests not related to any monitored workflow go directly to the application, without triggering a monitor query.

Below, we present two motivating examples that illustrate the configurations $\langle\mathcal{P},\mathcal{C},\mathcal{I}\rangle$, $\langle\mathcal{C},\mathcal{I}\rangle$ (first example), and $\langle\mathcal{I}\rangle$ (second example).

2.1 Example 1 - Enforcing constraints

Dolibarr[4] is an open-source ERP web application that implements a business process similar to the one shown at the top of Figure 2 (in BPMN) to manage customer invoices.

The process contains 6 tasks (depicted by rounded boxes). Tasks $t1$ to $t4$ must be performed in sequence (as indicated by the solid arrows), while either $t5$, $t6$ or neither are performed last (as indicated by the diamond-shaped gateway). Dolibarr implements each of the tasks shown in Figure 2. An authorization policy, control-flow, and possible data-flow invariants are implemented in an ad-hoc way, whose correctness is hard to verify, which may lead to vulnerabilities. The authorization policy provided by the application has a granularity of permissions that does not match the user-task assignment we support (there is no specific permission to, e.g., re-open an invoice). Authorization constraints are not supported. As a result, it is not trivial to prevent a malicious user from creating and validating a customer invoice (SoD between $t1$ and $t2$) or inserting and validating a payment (SoD between $t3$ and $t4$), which would allow him to, e.g., close invoices with an incorrect payment.

A user who wants to securely deploy this application can use Aegis to generate a $\langle\mathcal{P},\mathcal{C},\mathcal{I}\rangle$ monitor to enforce control-flow integrity, ensuring that all the steps in the customer invoice process are performed in the correct order; an authorization policy, ensuring that only authorized users can execute each task; and the SoD constraints described above, to avoid frauds. If the user prefers to leave authorization enforcement to the application, a $\langle\mathcal{C},\mathcal{I}\rangle$ monitor could be generated to only add support for constraints and integrity.

[4]http://www.dolibarr.org/

To generate a monitor for the invoicing process, without impacting other parts of the application, the user starts by collecting traces simulating users performing the process. Some HTTP traces representing these executions are (for the sake of readability, we show only simplified URLs, but headers and body are also part of the traces):

$\tau_1 =$ {`/invoice?action=create&value=10&prod=abc`, `/invoice/validate?id=1`, `/invoice/pay/create?id=1&value=10`, `/invoice/pay/validate?id=1`},

$\tau_2 =$ {`/invoice?action=create&value=20&prod=def`, `/invoice/validate?id=2`, `/invoice/pay/create?id=2&value=20`, `/invoice/pay/validate?id=2`, `POST /invoice/send BODY id=2`},

$\tau_3 =$ {`/invoice?action=create&value=30&prod=ghi&prod2=jkl`, `/invoice/validate?id=3`, `/invoice/pay/create?id=3&value=30`, `/invoice/pay/validate?id=3`, `/invoice/reopen?id=3`}.

Each trace τ_i represents one possible execution of the invoicing process and each request represents one task. The first four requests in each trace are essentially the same, but with different parameter values (e.g., `id` is 1 in τ_1, 2 in τ_2, and 3 in τ_3). They represent tasks $t1$, $t2$, $t3$, and $t4$. τ_1 is an example of the branch where only the first four tasks are executed, while $t5$ is executed after $t4$ in τ_2, and $t6$ is executed after $t4$ in τ_3. The traces are automatically analyzed to extract data-flow properties, annotated and aggregated into an event log, sent to a process mining tool and the resulting Petri net labeled with a task-to-URL map (**Step 1**). Figure 2 shows, at the bottom, the Petri net obtained from the process mining tool (ignore for a moment the dashed lines). The tasks in the net are labeled as t_i, with the following task-to-URL map:

t_1 :	`/invoice?action=create&value=<<I>>&prod=<<DC>>`,
t_2 :	`/invoice/validate?id=<<IID>>`,
t_3 :	`/invoice/pay/create?id=<<IID>>&value=<<I>>`,
t_4 :	`/invoice/pay/validate?id=<<IID>>`,
t_5 :	`POST /invoice/send BODY id=<<IID>>`,
t_6 :	`/invoice/reopen?id=<<IID>>`

Data-flow properties are represented by annotations on the URLs. The `<<IID>>` (instance identifier) annotation is applied to the elements used to bind all the requests to the same instance of a workflow, in this case the `id` parameter. The `<<I>>` (invariant) annotation is applied to values that should not change during the workflow, in this example the `value` of the invoice in t_1 should be the same as the `value` of the payment in t_2. The `<<DC>>` ("don't care") annotation is applied to parameters that should be present in the request to help identify it as a unique action, but whose values are irrelevant. The parameter `prod2`, which is present in the request of $t1$ only in τ_3, is dropped in the task-to-URL map because it is considered optional, i.e. a trace may represent an invoice with one or more products, so only the first `prod` parameter needs to be present.

The user then specifies the constraints that must be enforced, shown as dashed lines labeled by $\neq$ (abbreviating a SoD constraint) in Figure 2. The model is used to synthesize a monitor (**Step 2**), which is composed of a set of SQL queries encoding the fact that to perform a task t, all the predecessor tasks (according to the control-flow of the model) must have been executed, there must be an authorized user u who has not performed any conflicting tasks, and there must be other users capable of executing the remaining tasks without violating the policy and the constraints.

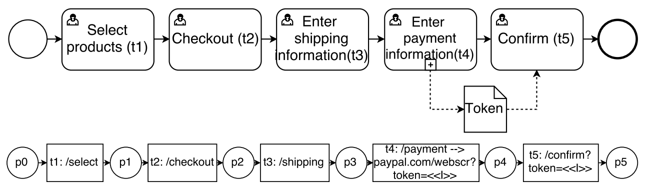
Select products (t1)
Checkout (t2)
Enter shipping information(t3)
Enter payment information(t4)
Confirm (t5)
Token
p0
t1: /select
p1
t2: /checkout
p2
t3: /shipping
p3
t4: /payment --> paypal.com/webscr? token=<<I>>
p4
t5: /confirm? token=<<I>>
p5

Figure 3: Checkout process in BPMN (top) and as a Petri net (bottom)

At run-time (**Step 3**), in the $\langle\mathcal{P},\mathcal{C},\mathcal{I}\rangle$ configuration, a policy is specified as a task-user assignment, e.g., $TA = \{(u1,t1),(u1,t2),(u2,t2),(u3,t3),(u4,t3),(u4,t4),(u5,t5),(u6,t6)\}$, where $(u,t) \in TA$ means that u is authorized to execute t. The assignment is stored in the database, and a reverse proxy is instantiated with the synthesized monitor. The proxy is capable of receiving a request such as `GET /invoice/validate?id=5` with the header `Cookie: sid=abcd1234` and identifying that it refers to task $t2$ of instance 5 of the invoicing process being performed by user $u2$ (whose cookie `sid` has been stored during login). It then queries the monitor and, assuming that $u1$ has previously executed $t1$ and $t2$ has not yet been executed, the SQL query is satisfied and the request is granted. On the other hand, a request can be blocked in several cases, such as if $u3$ tries to execute $t2$ ($(u3,t2) \notin TA$), if $u1$ tries to execute $t2$ (SoD), if any user tries to execute $t3$ before $t2$, or if any user issues a request for $t3$ with a `value` different from the one sent for $t1$.

To solve the WSP, regardless of the execution history, any request of $u4$ to execute $t3$ should be blocked. Granting that request would mean that the only user authorized to execute $t4$ has already executed $t3$, while both tasks are in SoD. Therefore, any execution where $u4$ performs $t3$ would either not terminate or terminate with the violation of some constraint or policy. This tension between business compliance and business continuity should be resolved and the synthesized monitor avoids it by blocking requests that lead to undesired situations.

2.2 Example 2 - Mitigating vulnerabilities

TomatoCart[5] is a popular e-commerce application that implements the checkout process depicted on the top of Figure 3. It is composed of 5 tasks executed in sequence, where $t4$ is a sub-process that can be implemented in different ways, but must produce a data object representing a token issued by a trusted third party, that is read in $t5$.

This is an example of a multi-party web application [14], which implements the payment step by using a third-party such as PayPal. An execution of this workflow, using PayPal Express Checkout, involves three actors: a client C, a service provider SP implementing TomatoCart and a trusted third party TTP implementing the payment provider. The execution starts with the client browsing the SP, selecting some product ($t1$), requesting checkout ($t2$), and entering shipping information ($t3$). The SP then contacts the TTP and receives a token identifying the transaction (not shown in the workflow). The user is redirected to the TTP with the token ($t4$), completes the payment (again not shown in the Figure), and is redirected back to the SP passing the token, which is verified to complete the transaction ($t5$).

[5]http://www.tomatocart.com/

In version 1.1.7, TomatoCart had a vulnerability that allowed users to replay a token in $t5$ of a new transaction and shop for free [14]. This vulnerability was manually fixed in a later release of the application, but Aegis could have been used to mitigate it until a patch was available (or until the patch could be applied, which is not always trivial). To mitigate the replay vulnerability, we can generate a monitor in the configuration $\langle\mathcal{I}\rangle$, enforcing control-flow integrity and the data invariant that the token received in $t4$ is the same one sent in $t5$. An authorization policy and authorization constraints are not specified since every user can execute the steps in the checkout process and all steps are executed by the same user. Details of the communication between SP and TTP and between C and TTP are not shown in the workflow because the monitor only needs to enforce that no user can replace the token that has been sent to him/her. Although Aegis ignores some messages, many vulnerabilities in multi-party web applications can be mitigated this way [20].

To generate the monitor, we repeat the steps presented for Example 1. Below, there are some traces of the execution of the checkout process, again simplified for readability. Now the traces involve three parties, thus each request must be identified with its host.

$$\begin{aligned}\tau_1 = \{&\texttt{shop.com/select}, \texttt{shop.com/checkout}, \texttt{shop.com/shipping},\\ &\texttt{shop.com/payment -> paypal.com/webscr?token=abcd1234},\\ &\texttt{shop.com/confirm?token=abcd1234}\},\\ \tau_2 = \{&\texttt{shop.com/select}, \texttt{shop.com/checkout}, \texttt{shop.com/shipping},\\ &\texttt{shop.com/payment -> paypal.com/webscr?token=efgh5678},\\ &\texttt{shop.com/confirm?token=efgh5678}\}.\end{aligned}$$

Figure 3 shows the Petri net obtained for the checkout process, labeled directly with the URL of each task (where `->` represents a redirect). The invariant annotation `<<I>>` is applied to the `token` received from PayPal, specifying that its value must be the same in `/payment` and `/confirm`.

A monitor is synthesized as before, however with neither authorization policy nor constraints. Workflow instances can be identified by the user identifier, since each user has only one checkout process at any given time. At run-time, whenever a user tries to replay a token, the monitor blocks this request because the token sent in $t5$ is different from the one received in $t4$. If the user tries to bypass the monitor by skipping step $t4$ and sending the token directly in $t5$, the monitor blocks the request because of a control-flow violation.

3. DETAILS

An HTTP trace (or a web session) is a sequence $S = \{(u_1 : r_1, s_1), (u_2 : r_2, s_2), ..., (u_n : r_n, s_n)\}$ of pairs of web requests r_i issued by users u_i (which may or may not be all distinct) and responses s_i. Each web request or response is defined as $r_i = (\mathit{headers}, \mathit{body})$ and the information we derive from a request is a tuple $(\mathit{method}, \mathit{url}, P)$, where $\mathit{method} \in \{\text{GET, POST}\}$, url is the requested URL, and P is a set of parameters of the form (k, v), which can be in the URL (in GET requests), the body (in POST requests) or in the headers (e.g., cookies or *Location* in redirects). Data values passed as, e.g., JSON can be flattened to the same representation. The parameters in P represent the data values later annotated with data-flow properties.

A workflow $W(T,U)$ is a set of tasks ($t \in T$) endowed by execution constraints involving users ($u \in U$). A web application is composed of a set of workflows $\Psi = \{W_1(T_1,U_1), ..., W_n(T_n,U_n)\}$. We take as input sets of web sessions $WS_i = \{S_1, S_2, ..., S_n\}$ and infer from each WS_i a workflow $W_i(T_i, U_i)$, using an off-the-shelf process mining algorithm, and a set of data property labels L_i, using heuristics. We also take as input, optionally, sets of authorization constraints C_i. We then use a monitor synthesis procedure $\mathcal{MS}(W_i, L_i, C_i)$ that returns a monitor M_i. M_i is capable of answering requests of the form "can user u perform task t?"—encoded as $can\_do(u, t)$— with True iff the control-flow in W_i and the data-flow in L_i are respected, no authorization constraint in C_i is violated, the requesting user u is authorized by an authorization policy *TA* (specified at run-time), and the workflow can be executed until the end.

At run-time, a reverse proxy receives an incoming request $u : r$ and based on the information taken from it, tries to translate it into a query of the form $can\_do(u, t)$, for $u \in U_i$ and $t \in T_i$ of workflow $W_i(T_i, U_i)$, which can be answered by M_i. Attacks on the application at the level of web requests characterized as follows [10]: a request forgery is an extra request not foreseen in a workflow ($\{r_1, r_2, ..., r_{forged}, ..., r_n\}$); a workflow bypass is a missing request ($\{r_1, r_2, ..., r_{i-1}, r_{i+1}, ..., r_n\}$); a workflow violation is an attempt to either repeat a unique request ($\{r_1, r_2, ..., r_i, ..., r_i, ..., r_n\}$) or execute a request out of order ($\{r_1, r_2, ..., r_{i+1}, ..., r_i, ..., r_n\}$); and authorization violations happen when a request is issued by a user who is not entitled to do so by the policy or when, for two tasks t_1 and t_2 in SoD, a user who previously issued a request r_1 to execute t_1, issues a new request r_2 to execute t_2. The monitor can mitigate these attacks because they do not comply with the expected workflow (naturally, they are only mitigated in the parts of the application covered by the inferred model).

3.1 Step 1 - Model inference

The goal of Step 1 is not to produce an accurate model of the whole application, but workflow models containing only security-relevant actions. These are the requests related to workflow tasks, whose execution should be controlled by the monitor. The definition of what is relevant varies from application to application, but besides the usual noise in HTTP traces (e.g., loading images and other resources), any request that leaves the application state unchanged should be filtered out. Such requests are called navigation events, as opposed to system-interaction events [13]. Not every system-interaction event should be controlled by the monitor (this should be decided by the user). However, discarding navigation events is crucial to keep the inferred models to a reasonable size and to eliminate imprecision due to variations in the process when executed by different users. We assume that this treatment of the input traces is done before Aegis is invoked. It can be done manually, but there are automated techniques to detect state changing requests [6, 13]. Such techniques are usually embedded in crawlers to obtain a model of the entire application. Applying just state-change detection to traces of a single workflow may have sub-optimal results (this evaluation is left to future work).

Since some URLs in an application can take different parameters and different values for these parameters, while still representing the same action, and since we apply differential analysis to identify data-flow properties, we need at least two different traces as input, each containing a possible value for each of the parameters (including their presence and absence). The input traces should also represent all the possible execution paths of the process (control-flow). The number of input traces required for a precise model depends on the number of control-flow branches in the workflow being analyzed, as well as the diversity of the traces. Related works use, e.g., four traces as input [20] or traces with specific requirements for each of the parties in the process [14]. At least two login traces with distinct users must also be present, so that cookies defining the user session identifier and parameters representing user names can be mined, to map requests to concrete users at run-time. From the set of HTTP traces, we extract three artifacts: a workflow model, a task-to-URL map, and a set of data properties.

Workflow model and map. A workflow model is automatically obtained from a process mining tool. There are many well-known process mining algorithms and a simple example is the α-algorithm [17]. It mines workflow nets by recording all the events in a log and detecting relations between them, such as sequence, exclusive, and parallel executions. In the traces used in Example 1, it is possible to see that $t1$ always precedes $t2$ and $t2$ never precedes $t1$, so the algorithm infers a causal dependency between them and adds a place connecting transitions $t1$ and $t2$ in the output net (place $p1$ in Figure 2). It is also possible to see that $t4$ precedes $t5$, $t4$ precedes $t6$, and $t5$ and $t6$ never happen in the same trace, thus the algorithm creates a place after $t4$ that branches the execution ($p4$ in Figure 2). Since the input traces contain only relevant URLs and each unique URL becomes a transition after process mining, the task-to-URL map is trivial to obtain.

Data-flow properties. We use five annotations, namely *constant*, *don't care*, *invariant*, *instance identifier*, and *user identifier*, which are used for three goals. *Constants* and *don't cares* are used to restrict and generalize, respectively, the input traces by fixing or hiding given values that are used to match incoming requests at run-time. A *user identifier* is used to detect the user issuing a request and an *instance identifier* to detect the workflow instance that the request is related to, since several instances of the same workflow may be running at the same time and they may have different execution histories. *Invariants* indicate values that should not be modified during a workflow instance execution.

Data-flow properties are obtained by using differential analysis, i.e. comparing the differences in the data values between traces, as is done in related work (e.g., [20, 14]). For each trace, the analysis compares the values of all parameters in each request in relation to (i) the same parameter in other requests of the same trace, (ii) the same parameter in other traces, (iii) other parameters in the same trace, and (iv) other parameter in other traces. Aegis does not apply syntactic annotation (as, e.g., [14]) to identify the data type of each parameter, and does not try to discover possible values or intervals for data elements, because it does not enforce particular values that were seen in the traces (except for *constants*). Below, we describe the differential analysis used to identify each kind of data-flow property.

Let *WS* be the set of traces τ_i used for analysis, each τ_i be composed of requests r_j and responses s_j, and each request or response have a set P of parameters (k, v). Considering the same request r_j in every trace $\tau \in WS$, if a parameter (k, v) appears in only a strict subset $\tau' \subset \tau$ of the traces, it is considered optional and ignored, i.e. dropped from the URL in the labeling function L. *Constants* are parameters that are present in every trace $\tau \in WS$ for the same URL of a request r_j and whose key k and value v never change. An example is the parameter `action=create`, which is in $t1$ of traces τ_1, τ_2, and τ_3 in Example 1. *Don't cares* are parameters that appear in every trace $\tau \in WS$ for the same URL of a request r_j and whose key k remains constant, but whose value v is different in at least one of the requests. One example is `prod=abc`, `prod=def` and `prod=ghi` in $t1$ of Example 1 annotated as `prod=<<DC>>`. An *instance identifier* is a key k whose value v is present in every request r of a trace τ, with different v's in every trace. In Example 1, the parameter `id` is an instance identifier, since it has the value 1 in every request of τ_1, the value 2 in every request of τ_2, and the value 3 in every request of τ_3. Notice that what must remain constant is the value and not the key, so it is possible to have an instance identifier called, e.g., `id` in one request and `iid` in another request. A *user identifier* is a parameter that comes from a response issued by the server, is stored in a cookie, sent in every request of a trace and whose value changes in every trace in WS. In Example 1, only URLs are shown in the traces, but the cookie `sid` is sent with every request, as can be seen towards the end of the example. *Invariants* are values v that remain constant during a trace, change between traces in WS and are not present in every request of a trace (as opposed to instance identifiers). Two examples are the `value` parameter in $t1$ and $t3$ in Example 1 and the `token` in $t4$ and $t5$ of Example 2. Like instance identifiers, invariant *values* should not change, but their *keys* might, so that an invariant can be called, e.g., `price` in one request and `amount` in another. There may be many invariants in a workflow, so they are annotated as `<<I_1>>`, ..., `<<I_n>>` for run-time enforcement.

Step 1 outputs a tuple (PN, L), where PN is a Petri net and L is a labeling function that associates to each transition in the net a URL annotated with the identified data properties. Although (PN, L) is obtained automatically, it can be edited by a user before being sent for monitor synthesis. Control-flow constraints can be changed by graphically adding or removing places or transitions in the Petri net (or tasks and gateways in BPMN), while data properties can be modified by adding or removing annotations on the URLs.

3.2 Step 2 - Monitor synthesis

Step 2 takes as input the (PN, L) from Step 1 and, optionally, augments it with security properties given by the user. As an example, the user can specify authorization constraints $\text{SoD}(tx, ty)$ indicating that tasks tx and ty must be executed by different users. The user must also indicate whether the monitor should enforce an authorization policy to be specified later.

Security properties specification. All behaviors of the web application that satisfy the specified security properties are represented by the executions of a symbolic transition system $S = (V, Tr)$, where V is a set of state variables and Tr is a set of transitions. In general, each workflow task corresponds to one transition. Each transition has a condition and an update part. The conditions specify the constraints that must be satisfied for a task to be executed (e.g., control-flow, data-flow and authorization) and are expressed based on the variables in V, therefore encoding the security properties. The update represents the effect of executing the task (changing the values of the variables in V). See [2] for more details on the variables and transitions in S.

Monitor synthesis. S is fed to a symbolic model checker, which computes a reachability graph RG representing all possible executions of the workflow by a set of symbolic users. A procedure to compute this graph, based on backward reachability, is described in [2]. RG is a directed graph whose edges are labeled by task-user pairs in which users are symbolically represented by variables and whose nodes are labeled by a symbolic representation (namely, a formula of first-order logic) of the set of states from which it is possible to reach a state in which the workflow successfully terminates. A Datalog [4] program M is automatically derived from RG by generating a clause of the form $can\_do(u, t) \leftarrow \beta_n$ for each node n in the graph (β_n is the formula labeling n). M is then translated to SQL [16]. The SQL program is capable of answering—after possibly being instantiated with a concrete authorization policy—user requests to execute tasks in a workflow in such a way that the authorization and execution constraints are not violated, the authorization policy is respected and termination of the workflow is guaranteed, thus enforcing the specified security properties and solving the run-time WSP.

Step 2 outputs a tuple (M, L), where M is the monitor generated from RG and L is the labeling function, which now maps from transitions in S to annotated HTTP requests.

3.3 Step 3 - Run-time monitoring

Step 3 takes as input (M, L) and, if previously specified, an authorization policy TA used to populate a database queried by M, resulting in a concrete monitor.

A reverse proxy intercepts all incoming requests to the application and decides, for each request, whether it is part of a workflow or not. To do so, it tries to match the URL and parameters in the request to annotated URLs and parameters stored in L, taking into account the constant, ignored and don't care values. If there is no match, the proxy forwards the request to the application, as it is not part of any workflow. If there is a match, the proxy associates the request to a task t of a workflow $W(T, U)$ and checks the annotated URL for `<<IID>>` and `<<UID>>` values, extracting the instance i and the user u. The user identifier is a cookie value that must be mapped to a user name in the policy. This is done by capturing login actions, storing the cookies issued to each user, and later retrieving the user names based on the cookie.

To enforce data invariants, when the proxy receives a request for the first URL containing the annotation `inv=<<I_i>>`, it stores the value of the parameter `inv` as v_i. When any subsequent task containing `<<I_i>>` is accessed, the value of the incoming annotated parameter `inc` is compared to the stored value ($v_i =$ `inc`). In these requests, the monitor query $can\_do(u, t) \leftarrow \beta$ is dynamically conjoined with the data invariant condition, becoming $can\_do(u, t) \leftarrow \beta \wedge v_i =$ `inc`. Finally, the proxy issues a request $can\_do(u, t)$ to the monitor of instance i of W and acts based on its response by either forwarding the request or dropping it.

4. EVALUATION

AEGIS was implemented in Python 2.7. We capture execution traces using ZAP, extract data properties from them, aggregate them into an XES log file and use ProM [17] to mine the Petri net. Monitor synthesis is implemented as in [2]. We instantiate mitmproxy[6] with the generated monitor script that intercepts requests and responses, performs URL matching, queries a MySQL database—which stores

[6]https://mitmproxy.org/

the authorization policies—by using the synthesized queries, and either forwards or drops the request. The proxy also supports HTTPS connections.

4.1 Experimental setup

We tested AEGIS on the ten open-source applications shown in Table 1, synthesizing monitors in the configurations $\langle \mathcal{P}, \mathcal{C}, \mathcal{I} \rangle$ and $\langle \mathcal{I} \rangle$. #1-4 are ERP platforms, #5-6 are e-health applications and #7-10 are e-commerce applications. Column *Application* contains the name of each application; *Language* shows the language in which it was developed; *Params* describes the predominant method used for parameter passing (although an application can use several methods) and *Downloads* reports the number of downloads (#1-6) or public installations (#7-10).

The different languages show the versatility of the black-box approach, which has to be tailored to support each parameter passing method. The number of downloads and installations is a measure of the popularity of the applications and comes from official repositories (#2, 3, 5, and 6), data in the web page of the project (#1 and 4), or related work [12] (#7-10). The number of actual deployments for #1-6 is not available as they are usually internal to an organization and not indexed by search engines.

We pre-configured the applications using demo data and captured four execution traces for each workflow and two login traces for each application. To compare AEGIS in different ERP applications, we used workflows offered by all of them: *Purchase order* (PO), *Sales order* (SO), *Purchase invoice* (PI), and *Sales invoice* (SI). They are slightly different in each application, varying from 4 to 6 tasks, usually with a gateway defining 2 to 3 alternative execution branches. Figure 4 shows at the top the patient visit workflow mined from OpenEMR (where the lines labeled by = are BoD constraints) and at the bottom the lab analysis workflow mined from BikaLIMS. In these 6 applications, we added the authorization constraints and specified policies with 10 users assigned to each task, generating $\langle \mathcal{P}, \mathcal{C}, \mathcal{I} \rangle$ monitors.

The workflows for e-commerce applications are similar to the one shown in Figure 3. For these applications, we use the $\langle \mathcal{I} \rangle$ configuration, thus neither constraints nor authorization policies were defined. Applications 7 and 8 have a vulnerability allowing the replay of tokens (CVE-2012-4934). Applications 9 and 10 allow an attacker to tamper with a parameter that indicates who should receive the payment for a transaction (CVE-2012-2991).

All applications were deployed as Docker containers and the tests as Selenium scripts, which allows us to automatically test the applications with and without monitoring.

Table 1: Applications used in the experiments

#	Application	Language	Params	Downloads
1	Odoo	Python	JSON	2M
2	Dolibarr	PHP	GET	850k
3	WebERP	PHP	GET	617k
4	ERPNext	Python	JSON	25k
5	OpenEMR	PHP	GET	382k
6	BikaLIMS	Python	REST	111k
7	OpenCart 1.5.3.1	PHP	GET	9M
8	TomatoCart 1.1.7	PHP	GET	119k
9	osCommerce 2.3.1	PHP	GET	80k
10	AbanteCart 1.0.4	PHP	GET	21k

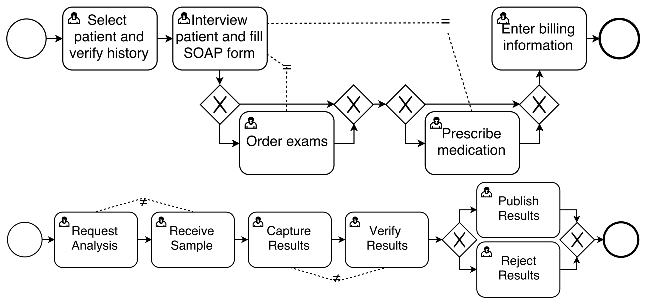

Figure 4: Workflows from OpenEMR (top) and BikaLIMS (bottom)

The experiments ran on a laptop with a 1.3GHz dual-core processor and 8GB of RAM.

4.2 Results

The enforcement of security properties and mitigation of vulnerabilities was successful in all applications; this was confirmed by manual inspection. In applications 1-6, we tried the attacks described in Section 3 (workflow bypass, workflow violations, and authorization violations). The monitor was able to block situations such as the same user executing an entire workflow (SoD violation), and users trying to access tasks that were not assigned to them. In applications 7-10, we tried to exploit the vulnerabilities described above. In applications 7-8, the attacks were unsuccessful because `token` was detected as an invariant and automatically enforced. In applications 9-10, the `PayeeId` parameter was detected as a constant, since every trace in the input was related to the same shop (constants are not enforced, only used to match URLs). We then edited the inferred model by annotating `PayeeId` with `<<I>>`, so that requests with any value of `PayeeId` are controlled by the monitor, and used invariant enforcement with a constant, instead of with the first received value, to check that in every request containing `PayeeId`, its value is equal to the one obtained in the traces.

Table 2 shows the performance of model inference, monitor synthesis, and run-time enforcement. Column *App.* shows the application under test (and the specific workflow tested for ERP applications); *Synth.* shows the time to infer a model from the captured traces and synthesize a monitor for each workflow. *Orig.* reports the time between receiving a request and sending a response with no monitor; *Query* reports the time for the monitor to answer a query (ignoring the time to invoke the script, translate an incoming request to a monitor query, forward the request, etc); *Aegis* reports the time of a response with the monitor script (the time taken by the application, plus translation time, plus querying); and *Overh.* shows the overhead incurred by the use of the monitor (the difference between *Aegis* and *Orig.*). The time in column *Query* varies with the size of a workflow and the number of users and constraints, as reported in [2].

The overhead varied from 8ms to 84ms, with a median of 13.5ms, out of which less than 10ms in most cases is spent in querying the monitor. The variability is due to the complexity of the workflows and the time to translate a request. For instance, Odoo and ERPNext have a large overhead because of the time to process JSON requests. Monitor synthesis is computationally much more expensive, but it is run only once for each workflow and has been shown to be scalable [2].

Table 2: Monitoring overhead

App.	Synth.	Orig.	Query	Aegis	Overh.
Odoo PO	21.3 s	112 ms	6 ms	132 ms	20 ms
Odoo SO	22.4 s	170 ms	7 ms	213 ms	43 ms
Odoo PI	14.3 s	174 ms	7 ms	213 ms	39 ms
Odoo SI	17.9 s	104 ms	7 ms	116 ms	12 ms
Dolibarr PO	14.2 s	93 ms	5 ms	103 ms	10 ms
Dolibarr SO	14.3 s	92 ms	4 ms	104 ms	12 ms
Dolibarr PI	13.2 s	89 ms	5 ms	97 ms	8 ms
Dolibarr SI	14.7 s	90 ms	5 ms	105 ms	15 ms
WebERP PO	20 s	51 ms	6 ms	59 ms	8 ms
WebERP SO	21.1 s	50 ms	5 ms	57 ms	7 ms
WebERP PI	18.3 s	30 ms	6 ms	37 ms	7 ms
WebERP SI	19.5 s	32 ms	4 ms	39 ms	7 ms
ERPNext PO	13.3 s	222 ms	7 ms	251 ms	29 ms
ERPNext SO	12.9 s	327 ms	14 ms	411 ms	84 ms
ERPNext PI	15.9 s	263 ms	10 ms	327 ms	64 ms
ERPNext SI	13.7 s	272 ms	13 ms	318 ms	46 ms
OpenEMR	19.1 s	95 ms	7 ms	112 ms	17 ms
BikaLIMS	31.2 s	306 ms	7 ms	326 ms	20 ms
OpenCart	19.1 s	65 ms	6 ms	77 ms	12 ms
TomatoCart	15.8 s	63 ms	4 ms	71 ms	8 ms
osCommerce	22.2 s	79 ms	7 ms	95 ms	16 ms
AbanteCart	19.8 s	117 ms	8 ms	127 ms	10 ms

5. CONCLUSION

We have described and evaluated Aegis, a tool to enforce authorization policies and constraints, control- and data-flow integrity, and ensure the satisfiability of web applications. Our experiments show the practical viability of our approach in enforcing the desired properties and mitigating related vulnerabilities with a small performance overhead.

Related work. Many works studied authorization, control- and data-flow integrity (separately) in web applications, and mitigation of related vulnerabilities, e.g., [1, 5, 3]. These approaches are white-box, whereas Aegis is black-box. Web application workflow models have been used to find vulnerabilities by identifying behavioral patterns from execution traces [12], but Aegis is focused on enforcement, so the techniques are complementary. Identification of data properties has been used in [20, 12, 14] with different goals. The enforcement of authorization constraints for collaborative web applications was studied in [8, 7], but there was no discussion about workflow satisfiability, and the evaluation was limited to prototypes. The closest related works are BLOCK [9] and InteGuard [20]. Both use a reverse proxy, construct policies using invariants from network traces, and rely on manual identification of critical requests. InteGuard is tailored for multi-party application integration, where most tasks are not performed by humans and workflows must be executed from beginning to end in one shot. Neither tool enforces authorization policies nor constraints.

Future work. We intend to test Aegis in more real-world applications and to explore monitor inlining by embedding synthesized monitors into the applications.

6. REFERENCES

[1] D. Balzarotti, M. Cova, V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of web-based applications. In *Proc. of CCS*, 2007.

[2] C. Bertolissi, D. R. dos Santos, and S. Ranise. Automated synthesis of run-time monitors to enforce authorization policies in business processes. In *Proc. of ASIACCS*, 2015.

[3] B. Braun, P. Gemein, H.P. Reiser, and J. Posegga. Control-flow integrity in web applications. In *Proc. of ESSoS*, 2013.

[4] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *TKDE*, 1(1):146–166, 1989.

[5] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proc. of RAID*, 2007.

[6] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proc. of USENIX Sec.*, 2012.

[7] P. Gaubatz, W. Hummer, U. Zdun, and M. Strembeck. Enforcing entailment constraints in offline editing scenarios for real-time collaborative web documents. In *Proc. of SAC*, 2014.

[8] P. Gaubatz and U. Zdun. Supporting entailment constraints in the context of collaborative web applications. In *Proc. of SAC*, 2013.

[9] X. Li and Y. Xue. Block: a black-box approach for detection of state violation attacks towards web applications. In *Proc. of ACSAC*, 2011.

[10] X. Li, Y. Xue, and B. Malin. Detecting anomalous user behaviors in workflow-driven web applications. In *Proc. of SRDS*, 2012.

[11] T. Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[12] G. Pellegrino and D. Balzarotti. Toward black-box detection of logic flaws in web applications. In *Proc. of NDSS*, 2014.

[13] M. Schur, A. Roth, and A. Zeller. Mining workflow models from web applications. *TSE*, 41(12):1184–1201, 2015.

[14] A. Sudhodanan, A. Armando, L. Compagna, and R. Carbone. Attack patterns for black-box security testing of multi-party web applications. In *Proc. of NDSS*, 2016.

[15] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *Proc. of USENIX Sec.*, 2011.

[16] G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory Pract. Log. Program.*, 8(2):129–165, 2008.

[17] W.M.P. van der Aalst. *Process Mining*. Springer, 2011.

[18] Q. Wang and N. Li. Satisfiability and resiliency in workflow authorization systems. *TISSEC*, 13(4):40:1–40:35, 2010.

[19] M. Weske. *Business Process Management*. Springer, 2007.

[20] L. Xing, Y. Chen, X. Wang, and S. Chen. Integuard: Toward automatic protection of third-party web service integrations. In *Proc. of NDSS*, 2013.