CliCap- Developer Documentation

Hoang-Duong Nguyen, Tobias Reinhard

July 29, 2015

Contents

Chapter 1

Introduction

1.1 Motivation

Over the last fifteen years online services in general and web applications in particular have become an inevitable part of our daily life. We use web shops to order books or laptops and even open the web browser to order pizza for lunch. Nevertheless, the increasing popularity of online shopping has lead to a wide range of attacks and attackers targeting web shops and all kinds of web services. Thus it is crucial to develop countermeasures protecting against such attackers. Due to the nature of complex software systems, protecting mechanisms are usually only able to defense against a certain set of attacks. However, even securing a web shop against a small set of attacks takes a lot of time and the implementation of such security mechanisms is often bound to a concrete application.

Therefore, the demand for a dynamic protecting mechanism that can be adapted to protect a wide range of web applications against a large set of vulnerabilities has been getting higher, especially in the last few years. For this, Gay et al. have developed a tool for cooperative dynamic enforcement of distributed Java programs which has a rigorous theoretical fundamental [1][2]. In this project we attempt to extend this tool to secure E-commerce web applications which utilize web APIs to incorporate third-party services into their functionalities. This approach is implemented in form of the CliCap project and the idea behind is presented in the following section.

1.2 The CliCap Project

1.2.1 Visions

CliCap works as an adaptable enforcement mechanism to an existing web application with the purpose to protect against possible security flaws. It is adaptable in the sense that it can be configured to work with a wide range of web applications. Furthermore, it is dynamic in the sense that the encapsulated security policy can be setup in a modular way to be able to protect against new security flaws. Thus every time new security issues arise CliCap can be configured accordingly. At the same time for every such web application it leaves the (intended) functionality and the administration of the web application untouched. In addition, once setup, it does not require any administrative work as long as no new security- or network issues arise.

Moreover CliCap works as a distributed system whose nodes can be located on different physical machines. It is scalable in the sense that it offers the administrator the possibility to dynamically add and remove nodes from the system. Doing so, the system itself remains consistent, sound and transparent during and after the change of nodes. Thus, such a change does not affect the functionality of the secured web application, which implies that nodes can be dynamically adapted to meet the current requirements. For instance, in case a massive number of requests slows down the protection layer formed by CliCap and therefore affects the user experience of the web application, the administrator can solve this simply by adding further nodes which are located on different physical servers. On the other side the administrator can also react to a decreasing amount of requests by removing nodes. Thus she can control that at any time point CliCap consumes only an amount of resources that is appropriate to the current situation.

In the remaining chapters the words "node" and "server" are used interchangeable. However, one single physical server is able to host several CliCap *nodes.*

1.2.2 Functionality

In order to fulfill the requirements stated above CliCap captures every HTTP package the web application receives as well as every HTTP package sent by the web application. Additionally, CliCap includes an internal and configurable policy responsible for the categorization of packages into security threats and legal packages. Every captured package will be analyzed whether or not it complies with the policy. All legal packages are allowed to pass CliCap normally i.e., they are forwarded to their original destination which is either the web application or a clients. However, packages categorized as security threats are not allowed to pass CliCap. Instead they are dropped, the administrator is informed about the policy violation and a security warning is sent back to the client responsible for the security violation.

In our project we chose to secure TomatoCart 1.1.7 [3] against the *pay-for-less* attacks reported in [4]. But still, CliCap can be configured to work with other web application and to secure against further attacks that are, in principle, preventable with a black-box approach.

Project name. Our tool is an extension of CliSeAu invented by Gay et al. [2] such that the *Interceptor* and *Enforcer* are replaced by developing an *ICAP service*, hence the name CliCap.

Chapter 2

Technical Description

2.1 Architecture

The CliCap project consists of several components. Though a detailed and technical explanation of these modules will be given in the following subsections, an overview and the intuition behind these modules shall be presented first. For this figure 2.1 presents the high level architecture of the CliCap project and the communication of the several modules including a reverse proxy, the major- and minor nodes.

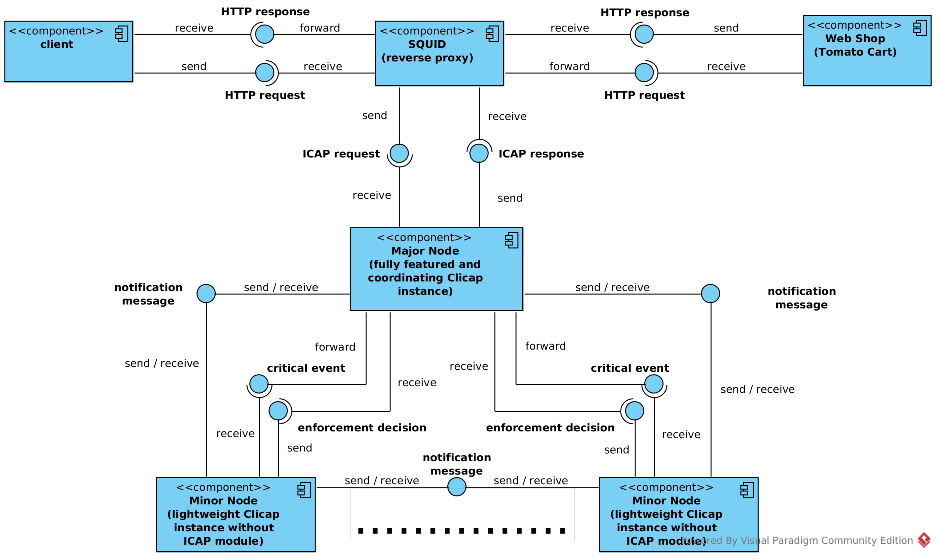


Figure 2.1: High Level view on the CliCap Project.

The basic concept is that clients and the web shop do not communicate directly with each other. Instead, all traffic (i.e. all HTTP packages) is intercepted by a reverse proxy, for which we use SQUID [5]. This proxy extracts relevant information and sends it to the major node in form of an ICAP request and awaits response in form of an ICAP response. This response will contain a decision whether the intercepted HTTP package shall be permitted or rejected.

In case this package is permitted, it will be forwarded to its intended destination (i.e. client or web shop). Otherwise it will be dropped and if it has been sent by the client, a HTTP package containing a security warning will be sent back to the client.

The major node is a fully featured CliCap instance i.e., a CliCap instance with all components presented in figure 2.2. In contrast to the major node, there are also minor nodes which are lightweight CliCap instances, i.e. CliCap instances without the ICAP submodule presented in figure 2.2 being instantiated. The major node is the gateway of the enforcement system and responsible for triggering the coordination between all enforcement units. It is the only CliCap instance that communicates directly with the reverse proxy server. As a consequence, the major node receives all ICAP requests and for each such request a responsible CliCap instance can be determined uniquely. In case this responsible unit is the major node itself, it decides weather to permit or reject the request and sends an enforcement response immediately to the proxy. Otherwise, it forwards the received ICAP request to the responsible minor node and awaits its response which is then forwarded to the proxy. Thus each minor node is responsible for the policy enforcement of a concrete subset of all possible ICAP requests. Furthermore, the set of minor nodes is not static as the administrator can add new (minor) nodes to and remove existing (minor) nodes from the setup. For this it is necessary that the minor nodes communicate with each other and exchange data because changes in the setup of minor nodes will also lead to changes in responsibilities regarding the critical HTTP packages.

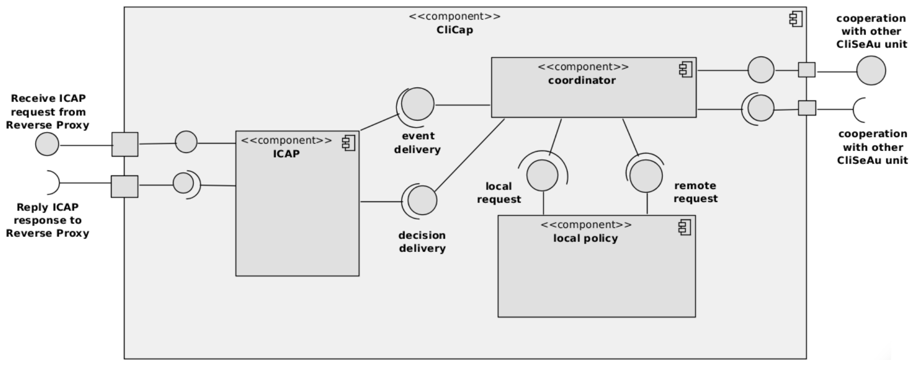
The following sections will describe more technical details of the modules.

2.2 Interface between ICAP server and CliSeAu

2.2.1 Use cases

This module mediates between the target system (in other words the HTTP communications between the web application and its clients) and the major CliSeAu unit. There are two classes of HTTP messages that are captured by the reverse proxy and forwarded to CliCap:

1. HTTP request sent from clients to TomatoCart
    - Requests that are determined as noncritical will be encapsulated into an ICAP response and sent back to the Reverse Proxy. It will be unpacked there and forwarded to the web application.
    - If the request is critical, ICAP service must wait for enforcement decision made by the major CliSeAu unit. After that it will continue its work as an enforcer according to the received enforcement decision.
2. HTTP response sent from TomatoCart back to clients
   All HTTP responses of the web application are allowed to be forwarded to the clients. Only the one that piggybacks a *token* generated by TomatoCart is captured before forwarded in order to extract the needed information required by the local policy of the major CliSeAu unit.



<<component>>
CliCap
<<component>>
coordinator
<<component>>
ICAP
<<component>>
local policy
Receive ICAP request from Reverse Proxy
Reply ICAP response to Reverse Proxy
event delivery
decision delivery
local request
remote request
cooperation with other CliSeAu unit
cooperation with other CliSeAu unit

Figure 2.2: Interaction between the components of the system

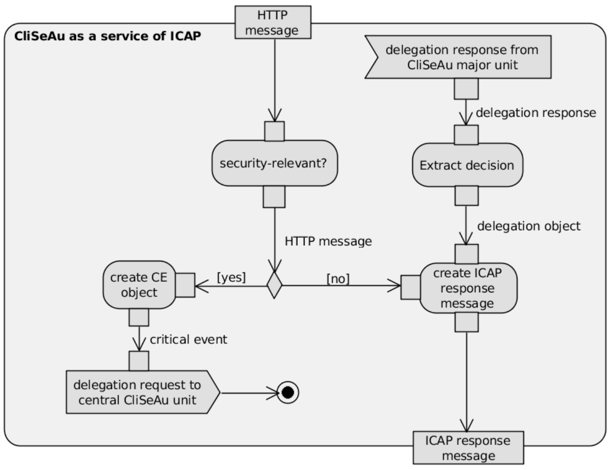


Figure 2.3: Service that ICAP deploys to coordinate with CliSeAu

2.2.2 Architecture

Two main parts of the interface are the ICAP service `icap.services.Cliseau` (figure 2.2) and the starter `cliseau.CliCap`. Basically the interceptor and enforcer are replaced by a reverse proxy (SQUID), ICAP and its *Cliseau* service (figure 2.3). The communications between components of the system are otherwise equivalent to those descibed in [1]. Connections between ICAP and CliSeAu coordinator are currently hard-coded as the ports and domains are specified in `cliseau.CliCap`.

2.2.3 Functional requirements

This module provides the local policy all the necessary information about the security-relevant events that occur in the monitored system. On the one hand, all HTTP messages relate to the events listed in 3.2 must be recognized and analyzed correctly. On the other hand, the critical event factory (`cliseau.central.IcapEventFactory`) must guarantee that correct abstract critical events are generated based on the received information about the HTTP messages. Furthermore, the ICAP service is able to send back a HTTP response directly to the client which will display a warning page if this user behaves illegally according to the security property specified by the security automaton. All in all the module supplies all materials that the local policy requires for making enforcement decision.

2.2.4 Design patterns and guidelines

Since the implemented service for ICAP server deploys CliSeAu to enforce security policies, it also uses the factory design pattern as in CliSeAu [2]. All the implementations follow the Java naming standard.

2.3 Major and Minor Nodes

2.3.1 Use Cases

The complete tool is developed to protect an online shop against attacks that exploit logical flaws of the shop software. The responsibility of the major node is to receive critical events (c.f. section 3.2) from the reverse proxy and send some decision for each event back stating whether the event shall be permitted or rejected. For each critical event a uniquely determined CliCap unit is responsible (either the major node itself or some minor node). In case the major node receives some event for which it is responsible, it makes the decision itself. Otherwise the event is forwarded to the responsible minor node which makes a decision and sends it back to the major node. After that the major node can send the decision back to the proxy.

To make a decision upon a concrete critical event, each node follows the policy defined in section 3.3.

However, these modules (i.e. major and minor nodes) can be used in any tool for the protection of online shops against pay-for-less attacks. For this it is necessary that the tool generates the needed critical events and that the major and minor node's policy is updated according to the specific trace of critical events produced by the new online shop.

2.3.2 Architecture

The major and minor nodes form a distributed system in which the major node works as a semi-central coordinator as it delegates critical events to the nodes that are responsible for the enforcement of the policy on this concrete event. Therefore the architecture of the major and minor nodes are almost identical. Both contain all necessary submodules to enforce the policy on a concrete set of critical events. The only difference is that the major node contains an ICAP module as described in section 2.2.

Module Architecture

In the following the architecture of the major node and the minor nodes will be presented. For this figure 2.2 presents the architecture of the major node. The architecture of the minor nodes is almost the same but without the ICAP module.

The major node is responsible for the decision making whether critical events shall be permitted or rejected. Therefore it offers an interface for the receiving of critical events as well as an interface for the returning of decisions. Internally the major node consists of three submodules the *ICAP* module, the *coordinator* and the *Local Policy*. The ICAP module is responsible for the receiving of ICAP requests from the proxy server and transforming them into critical events for the coordinator. Furthermore it also receives made decisions from the coordinator, transforms those into ICAP responses and sends them back to the proxy server. Details on ICAP and SQUID (the proxy server) can be found in section 2.2. Thus further descriptions will be omitted in this section.

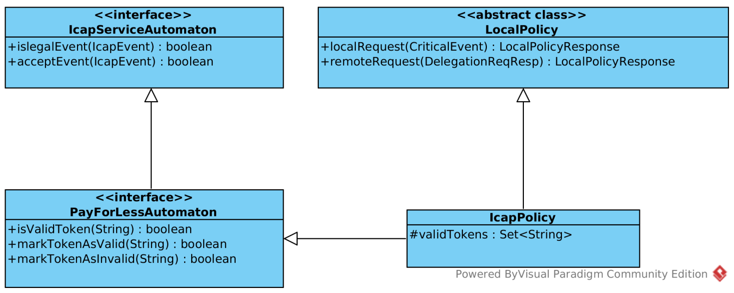
Local Policy implements the actual policy and therefore the decision making. It also contains an internal database to store all information contained in received critical events that might be necessary for the future decision making.

As the Local Policy module forms the instantiation of the security automaton defined in section 3.3, the internal data base is the instantiation of the automaton's state while the rest of the Local Policy module is the instantiation of the transition relation. Whether or not the automaton is able to make a transition does not only depend on the critical event but also on the current state. Furthermore, each transition might change the automaton's state i.e., each decision making upon a concrete critical event might change the data base's content.

Class Architecture

In the previous section the abstract architecture of the major and minor node modules has been presented. However, in this section the architecture of the major and minor node's implementation shall be presented i.e., the architecture on class level. A graphic representation of this architecture can be found in figure 2.4

Figure 2.4: Class Diagram of Major and Minor Node.



<<interface>>
IcapServiceAutomaton
+islegalEvent(IcapEvent) : boolean
+acceptEvent(IcapEvent) : boolean
<<abstract class>>
LocalPolicy
+localRequest(CriticalEvent) : LocalPolicyResponse
+remoteRequest(DelegationReqResp) : LocalPolicyResponse
<<interface>>
PayForLessAutomaton
+isValidToken(String) : boolean
+markTokenAsValid(String) : boolean
+markTokenAsInvalid(String) : boolean
IcapPolicy
#validTokens : Set<String>
Powered ByVisual Paradigm Community Edition

In the architecture on module level presented in the previous section the major and minor node consisted of three submodules, the ICAP module, the coordinator and the Local Policy. As stated before the ICAP module is not relevant for this section and the coordinator is part of the CliSeAu framework. The remaining part is the Local Policy module which is the instantiation of the security automaton defined in section 3.3. It implements the interfaces IcapServiceAutomaton and PayForLessAutomaton and extends the abstract class LocalPolicy from the CliSeAu framework. Furthermore it implements all abstract methods declared by IcapServiceAutomaton, PayForLessAutomaton and LocalPolicy that can be seen in figure 2.4.

2.3.3 Functional Requirement

From a functional perspective the major node requires a module providing critical events as input for the major node. Thus it requires some kind of event factory and coordinator that generate critical events and delegate them to the major node. On a class level the major node requires several independent classes and interfaces from the CliSeAu framework as well as from this project. It requires the classes IcapEvent, IcapEnforcementDecision and LocalPolicy from this project and the CliSeAu framework respectively. Furthermore it requires the interfaces CriticalEvent and DelegationReqResp from the CliSeAu framework. Therefore it also requires instantiations of these interfaces. (Note that IcapEvent is an instantiation of CriticalEvent.)

2.3.4 Context Diagram

The major node expects its input the critical events as instances of the class IcapEvent and returns its decisions in form of instances of the class IcapEnforcementDecision. In the current setup the major node receives the critical event from the interface between the Icap Server and CliSeAu. Therefore it receives the events basically from the Icap Server. The decisions are returned over this interface to the Icap Server.

2.3.5 Design Patterns and Guidelines

We defined two interfaces `IcapServiceAutomaton` and `PayForLessAutomaton`. The first one models the instantiation of a service or security automaton that works on events generated by the Icap Server. The second one models the instantiation of a security automaton that works on events generated by the Icap Server and that also intends to protect agains pay-for-less attacks. Both of these interfaces declare methods that should be implemented in the respective scenario. This allows the reuse of concepts on different abstraction levels, concepts we used for the implementation of the major node's main class IcapPolicy.

Furthermore we followed the official Java Naming Conversions which can be found following the url given below

http://www.oracle.com/technetwork/java/codeconventions-135099.html

2.4 Routing

A Chord-like routing algorithm [6] is used in our system in order to achieve decentralized coordinated enforcement. Data (here: tokens carried by critical events) is mapped to numeric identifiers. Each server has an unique ID that determines the range of data identifiers for which it is responsible. Only events that carry a token are considered in this mapping. For this we hash the piggybacked token and divide the result by the maximal number of server identifiers and take the remainder as the key identifying the corresponding critical event.

2.4.1 Chord Finger Table constructor

A shell script is used to trigger Service Automata instantiations (one of them encapsulates an ICAP server). This script can be generated by `scriptGen.java` (to do this please follow the instruction in the *README* file). The generator will print out a shell script which triggers desired servers which are configured according to the given information above. The configuration arguments of each server contain the finger table entries for this server.

2.4.2 Method for determining the next route

Our approach follows the algorithm presented in [6]: every time a node receives a requested key, this node first checks whether it is responsible for the given key. If not, it determine the next node on the routing path by looking up its finger table for the nearest predecessor of the key. Otherwise, the node is responsible for the given key and hence the routing path ends up here.

2.5 Network Scaling

The coordination network can be scaled out (i.e. adding new servers) or scaled in (i.e. removing servers) by the administrator. The scaling algorithms are those presented in [7], adapted to suit the message-based approach of CliCap.

2.5.1 Scale Out

When the administrator requests for adding a new node with the given ID, host and port, the following steps will be performed:

1. The major node generates a finger table for the new node with ID N by repeatedly querying for $N + 2^i$ for $0 \leq i < k$ (k is the maximal bit-length of server identifiers). The finger table might be incorrect at some fingers due to the fact that the network still does not know about the new node. Hence, the major node has to correct the collected finger table by traversing backwards it and replace the incorrect entries with the new node if needed.

2. The major node queries for the responsible node for the ID of the new node, in other words the new node's intermediate successor. After that they exchange information about the joining process and the major node can now instantiate the new node using received data.

3. After the new node is successfully instantiated, the successor is informed to transfer the corresponding enforcement data it maintains to the new node. After receiving this data, the newly joined server notifies its predecessor. Note that both the successor and the predecessor will update their pointer to the new node as soon as they are informed about the successful instantiation of that node. This fact is crucial for the correctness of the stabilization process which will be performed by the new node later on.

4. After receiving reply from the predecessor, the new node triggers the stabilization process, that is, updating finger tables of all server in the network

that should have the new node in them. To achieve this we follow the algorithm presented in [7]

2.5.2 Scaling In

After receiving scaling out request from the administrator which carries the ID N of the server to be removed, the following steps will be performed:

1. The major node signals N about its leaving process. Node N will then transfer all enforcement data it maintains to the successor and after that it will, until its departure, forward all further security-relevant requests to its successor. In other words, the leaving node acts as it is no more a member of the network.

2. The leaving node now triggers the stabilization of all finger tables maintained by nodes in the network that contain its ID. To achieve this we used an adapted version of the algorithm provided in [7]. The main difference with the original algorithm is how a finger table is updated.

3. After all finger tables are up-to-date, the leaving node notifies its predecessor and successor. These node then update their corresponding pointers and send back confirmations. Finally, N notifies the major node and leaves the network by terminates itself.

Race condition.
To achieve transparency and soundness of the enforcement mechanism during the scaling process, our approach guarantees the following properties: (1) once the responsibility has been "transfer" from node V to node W, V will forward all further corresponding requests to W (2) V is the first node that is knows about the joining of W, or W is the first node that is informed about the leaving of V (3) Before receiving any enforcement request from nodes other than V, W has already received the data from V.
All in all this ensures that the joining node (or the successor of the leaving node, respectively) will receive the up-to-date enforcement data from the successor (or the leaving node) and all further enforcement requests that require this responsibility will arrived in this node after it has successfully obtained all the data. However, our approach bases on two assumptions: (1) Transmission is reliable, that is, no message is lost, and (2) the order of messages sent out by a node to another node will remain unchanged by arrival.
By setting time-out for each communication, the system would work rightly without the first assumption. Though, without the second assumption race condition could occur such as the following scenarios:

- *Scaling out*: the joining node N has not yet received the data from the successor but an enforcement request forwarded by the successor after data is sent out arrives at N. Although this request should not be permitted but because N has not known about the token so it considers the token as valid and allow the critical event to occur. This could be solved by queuing all early in-coming requests by N.

- *Scaling in*: the successor S of the leaving node N has not yet received the data from N but an enforcement request forwarded by N after data is sent out arrives at S. The situation is the same as above: the request might not be permitted but because S has not known about the token so it considers the token as valid and allow the critical event to occur. This could also be solved by queuing all early in-coming requests by S.

Chapter 3

Formalization of System Properties

3.1 Attack

The attack that we protect against is classified as one of the *class 2* - attacks which make use of the *waypoints detour pattern* by removing a part of the navigation graph between two waypoints [4]. Therefore the propagation chains are reconstructed by fetching the missing data values from another user session (in this particular case the *token* and *payerID*). The details of this attack is as follows:

- The attacker creates a TomatoCart account. Fiddler is used to catch all the HTTP transactions during the shopping process.
- Attacker logs in her account, adds a cheap item into her shopping cart and checks out. She is then redirected to PayPal and there she logs in her PayPal account.
- After successfully being authorized by PayPal, attacker completes the payment by clicking on the button "Jetzt Bezahlen" (Now Paying), then she is redirected to TomatoCart and the legal shopping is successful. At this point the attacker does her work:
    1. Comes back to Fiddler and searches for the needed HTTP response which contains the $Token$ and $PayerID$ sent from PayPal, saves the respective redirecting link.
    2. Do a further shopping, this time chose some expensive items and click on "checkout".
    3. Now the attacker sends the old ($Token$, $PayerID$) directly to TomatoCart by navigating to the saved redirecting link from *step* 3. After that the shopping is successful without any payment which means the attacker must only pay for the cheap item in her first shopping.

3.2 Security-relevant events

According to the result of our careful analysis of HTTP transactions between the web application and its clients, we consider the following events as security-relevant (though only the 4th and 5th events must be considered in order to protect against the attack described above):

1. `LOGIN_C(sid,email)`
   Client with session ID *sid* and email address *email* tries to login.

2. `CONFIRM_ORDER_C(sid,order)`
   User with session ID *sid* clicks on "Confirm Order" on the Checkout-page. *order* is the body piggybacks by the respective HTTP request to be sent.

3. `TOKEN_ESTABLISH_C(sid)`
   User with session ID *sid* sends a HTTP request in order to receive a token.

4. `TOKEN_ESTABLISH_S(token)`
   The web application generates *token* and sends it back to some client.

5. `RECEIVE_PAYER_ID_C(sid,token,payerID)`
   The web application receives (*token,payerID*) piggybacked by a HTTP request sent by a lient with session ID *sid*.

6. `PROCESS_ORDER_C(sid)`
   The order of client with session ID *sid* is ready to be processed.

7. `SUCCESSFUL_ORDER_C(sid)`
   The order of client with session ID *sid* is successfully processed.

8. `LOGOUT_C(sid)`
   Client with session ID *sid* tries to logout.

Note: the suffix "`_C`" means the event is a HTTP request sent by the client and "`_S`" denotes a HTTP response sent by the web application to some client.

3.3 Security Automaton

In the following we define a security automaton that forbids Pay-for-less attacks targeting the TomatoCart 1.1.7 by forbidding the reuse of these tokens generated by the web application.
Let *Tokens* be an infinite countable set of tokens. Then the set of events E captured by the automaton is defined as

$$E := \{\mathit{TOKEN\_ESTABLISH\_S}(\mathit{token}),\ \mathit{RECEIVE\_PAYER\_ID\_C}(\mathit{sid},\mathit{token},\mathit{payerID}) \\ \mid \mathit{token} \in \mathit{Tokens}\}$$

Every event $\mathit{TOKEN\_ESTABLISH\_S}(\mathit{token})$ models a message from server to client containing a new token token that identifies this order and the check-out process. Every event $\mathit{RECEIVE\_PAYER\_ID\_C}(\mathit{sid},\mathit{token},\mathit{payerID})$ models a message from the client with session ID $\mathtt{sid}$ to the server confirming the payment of the order identified by $\mathtt{token}$ and the payer ID $\mathtt{payerID}$ sent by PayPal. The security automaton is defined as

$$(Q, \{q_0\}, E, \Delta)$$

In which:

$Q := \mathcal{P}(\mathit{Tokens})$
$q_0 := \emptyset$
$\Delta \subseteq Q \times E \times Q$ such that:

$$\Delta := \Delta_{\mathtt{TOKEN\_ESTABLISH\_S}} \ \cup \ \Delta_{\mathtt{RECEIVE\_PAYER\_ID\_C}} \quad \text{where}$$

$$\begin{aligned}\Delta_{\mathtt{TOKEN\_ESTABLISH\_S}} \quad &:= \{(\sigma, \mathtt{TOKEN\_ESTABLISH\_S(token)}, \sigma') \in Q \times E \times Q \\ &\quad \mid \ \mathtt{token} \notin \sigma \ \wedge \ \sigma' = \sigma \cup \{\mathtt{token}\}\}\end{aligned}$$

$$\begin{aligned}\Delta_{\mathtt{RECEIVE\_PAYER\_ID\_C}} \quad &:= \{(\sigma, \mathtt{RECEIVE\_PAYER\_ID\_C(sid, token, payerID)}, \sigma') \\ &\quad \in Q \times E \times Q \ \mid \ \mathtt{token} \in \sigma \ \wedge \ \sigma' = \sigma \setminus \{\mathtt{token}\}\}\end{aligned}$$

Intuitively for each $\mathtt{token}$ in a system state that is reachable from q_0 there is an event of the form $\mathtt{TOKEN\_ESTABLISH\_S(token)}$ but no event of the form $\mathtt{RECEIVE\_PAYER\_ID\_C(sid, token, payerID)}$ that has been captured.

3.4 Justification

3.4.1 Adequacy of security-relative events

The abstract events are adequate to model attacks of class 2 presented in [4] because these attacks exploit the logical flaw of TomatoCart 1.1.7 such that one can reuse the same *Token* and *PayerID* to complete an arbitrary number of additional fake transactions. This process is only bounded by the timeout set by PayPal on the token. The provided set of abstract events can capture all the tokens established by the web application and those used by the clients and hence is adequate.

3.4.2 Soundness and Transparency of protecting mechanism

The protecting mechanism makes use of an ICAP server to monitor all HTTP transactions between clients and the web application and enforces the security property specified by the automaton presented in the previous section. It is guaranteed to satisfy requirements on both soundness and transparency.

According to [8], an enforcement mechanism must ensure that all observable outputs obey the given property. In this case, the sequences of output events are traces that accepted by the security automaton specified above. On the one hand, $\Delta_{\mathtt{TOKEN\_ESTABLISH\_S}}$ ensures that the mechanism keeps track of all tokens that have been already generated by TomatoCart. On the other hand, $\Delta_{\mathtt{RECEIVE\_PAYER\_ID\_C}}$ guarantees that on every observed output sequence no token can be used for more than one legal shopping process. Hence, the set of traces accepted by the provided security automaton obeys the given requirement which is to protect against Pay-for-less attack presented in [4].

As tokens generated by TomatoCart are unique identifier of orders and a conventional user only receives her token and sends it back together with the *payerID* generated by PayPal once, all possible event sequences produced by legal shopping processes are accepted by the provided security automaton. Hence, if a user behaves legally, all HTTP requests sent from him will be forwarded to TomatoCart. In addition, the mechanism does allow all HTTP responses sent from the web application. All in all, every legitimate service usage is permitted by the provided protecting mechanism.

3.5 Limitation

Our mechanism is not able to protect against attacks of *class 3* presented in [4]. These attacks make use of the *waypoints detour pattern* by removing a part of the navigation graph between two waypoints. As a consequence, the propagation chains are reconstructed by fetching the missing data values from another user session. Hence, client identification is crucial for detecting such attacks. Though, after a careful analysis of the HTTP communication between *TomatoCart* and its clients, we conclude that mapping a *token* to its corresponding *session ID* is impossible (recall that the event `TOKEN_ESTABLISH_S(token)` does not contain *session ID* of the client who triggered the corresponding `TOKEN_ESTABLISH_C(sid)` event). In addition, due to the statelessness essence of HTTP protocol we can identify the clients only by their *session ID*. However, this number is stored in a cookie and hence can be easily modified by the clients. All in all there is no possibility to detect the reuse of token generated for other client by just observing HTTP communication. For this reason, our mechanism is not sufficient for protecting against *class 3 attacks*.

Chapter 4

Implementation

CliSeAu is a tool for hardening distributed Java programs and each CliSeAu unit consist of five components: a coordinator, a local policy, an interceptor, an enforcer and the target program [2]. In this project only the first two components are deployed as a service for the GreasySpoon ICAP server [9].

Assumptions.
For our implementation we rely on two assumptions:

1. The network is reliable i.e., no message is lost.
2. The order of messages sent from one node to another node is unchanged. Let m_1, m_2 be messages sent by some node X in the given order to some node Y, then by this assumption node Y receives m_1 before m_2.

4.1 Interface between ICAP server and CliSeAu

In order to build up an interface between ICAP server and CliSeAu we attempt to use another *control of policy* for ICAP server by developing a service that mediates between ICAP server and CliSeAu. Basically we just take "a half" of CliSeAu as a service for ICAP server instead of GreasySpoon. The CliSeAu parts used for the project are the local policy and coordinator. The interceptor, enforcer and target program to be encapsulated are cut off. Intuitively ICAP together with SQUID act as an interceptor and enforcer while CliSeAu plays the role of a local policy and coordinator between CliSeAu units.

SQUID is configured such that all HTTP requests and responses of the web application and clients are forwarded firstly to the ICAP server. This is exactly the task of an interceptor which intercept every event of the system and uses a "filter" to determine if an event if security-relevant ot not. This "filter" is implemented in the methods `getReqModCE` and `getRespModCE` in `icap.services.Cliseau`. Events that are considered as security-relevant are those defined in 3.2 .

ICAP server is configured to use the *Cliseau* service instead of *GreasySpoon*. This service is implemented such that every critical HTTP message will be sent to the ICAP server will be forwarded to the major CliSeAu unit. The service will then wait until it receives the respective decision from the local policy and after that either lets the message pass through SQUID or blocks it and replaces by a warning page. This means the newly implemented service for ICAP server *also acts as an enforcer* which executes decisions made by the local policy.

The interface between ICAP and CliSeAu involves class `IcapEventFactory` of the major CliSeAu unit. This class creates instances of `IcapEvent` according to the given data extracted from HTTP messages. Generated critical events contain all information that the local policy requires in order to make correct enforcement decisions. Data is transmitted internally through a communication channel created in `cliseau.clicap` which is also responsible for initializing ICAP service and major CliSeAu unit.

All in all the implemented interface provides a flexible framework for instantiating security policies that keep track of HTTP transactions between clients and a web application. In this particular case we replace CliSeAu's interceptor and enforcer with a newly created service for the ICAP server in order to take advantage of this protocol. The approach used here is also a flexible way of using CliSeAu.

4.2 Local Policy

1. Critical Events
   Critical events are instantiated in form of a class called `IcapEvent` that implements the CliSeAu interface `CriticalEvent`. Objects of this class are used to instantiate all the events described in section 3.2. Therefore it contains a field `type` of Type `IcapEventType` that indicates the event's type i.e. indicates if it is a *LOGIN_C*`(...,...)` or a *TOKEN_ESTABLISH_S*`(...)` event etc. `IcapEventType` is an enum that contains for each event type like *LOGIN_C* or `TOKEN_ESTABLISH_S` exactly one constant value. In order to instantiate all these events properly the class consists of one field for each type of parameter used by the events. Thus, it contains the fields*sid*, *email*, *token*, *order*, *payerID*.

2. Event Factory
   The event factory is instantiated in form of a class `IcapEventFactory` which implements CliSeAu's `CriticalEventFactory` interface. This class contains exactly one static method for each event type. These methods basically accept a HTTP package as argument (devided into header and body) and construct the corresponding critical event i.e. an `IcapEvent` object.

3. Enforcement Decision
   An enforcement decision is an instance of class `IcapEnforcementDecision` that implements CliSeAu's `EnforcementDecision` interface. The class contains an inner enum type `Decision` and a public field `decision` of type `Decision`. The enum `Decision` consists of the constants `PERMIT` and `REJECT`. Obviously these constants model the decision to permit and to reject an event, respectively.

4. Local Policy

   The local policy is instantiated in form of a class `IcapPolicy`. This class is an instantiation of the security automaton defined in section 3.3.

   More precisely we defined an interface `IcapServiceAutomaton` that should be implemented by every CliSeAu instantiation of an security or service automaton that accepts events instantiated by the class `IcapEvent`. This interface declares two methods

   ```
   boolean isLegalEvent(IcapEvent),
   boolean acceptEvent(IcapEvent)
   ```

   The first method checks if the automaton can accept the given event and make a transition in its current state. The second method checks the same and if a transition is possible, it also updates the automaton's state.

   Furthermore, we defined another interface `PayForLessAutomaton` that should be implemented by every CliSeAu instantiaton of an security automaton that accepts events instantiated by the class `IcapEvent` and that claims to prevent pay-for-less attacks. This interface extends the previous one and introduces the following methods

   ```
   boolean isValidToken(String),
   boolean markTokenAsValid(String),
   boolean markTokenAsInvalid(String).
   ```

   All those are auxiliary methods for the instantiation of the security automaton defined in section 3.3. For the explanation remember that each state of the security automaton is a set of tokens. The first method expects a token and states if it is contained in the state. The second one adds the given token to the state and the third one removes the given token from the state. So, methods two and three directly update the automaton's state. Therefore they should be invoked within the implementation of `acceptEvent(IcapEvent)`.

   The concrete instantiation of the security automaton defined in section 3.3 is, as stated to the beginning, the class `IcapPolicy`. This class implements the interface `PayForLessAutomaton` and contains a set of tokens which models the system state. The method's described above are implemented straight forward. Furthermore this class implements the CliSeAu policy for a centralized scenario. Therefore the method

   ```
   LocalPolicyResponse remoteRequest(DelegationReqResp)
   ```

   only consists of a dummy implementation and always returns null. In contrast to this, the method

   ```
   LocalPolicyResponNotese localRequest(CriticalEvent)
   ```

   is implemented properly. It throws an IllegalArgument exception on events that are not derived from `IcapEvent` and for every `IcapEvent` object it calls

   ```
   boolean acceptEvent(IcapEvent)
   ```

and returns a decision reflecting the result returned by this method. For every event acceptable in the current state `acceptEvent` returns true, otherwise false. Thus for every acceptable event, `localRequest` returns a permitting enforcement decision, otherwiNotese a rejecting one.

4.3 Scaling

4.3.1 Scaling Out

In the code the Scaling Out module described in section 2.5.1 is referred to as *Joining Protocol*. It is implemented in the class

```
cliseau.central.policy.scaling.JoiningProtocol
```

This class contains public, static methods for each step that has to be performed during the joining of a new node. Those steps are assumed to be called in the right order and before a scale out request is processed, the class is assumed to be initialized, i.e. the following method is assumed to be called before any other method is called

```
JoiningProtocol.init(LocalPolicy).
```

For all communications related to the Joining Protocol, the class

```
cliseau.central.delegation.IcapJoiningNotification
```

is used. This class contains several fields for the transported information as well as a `type` field relating the current message to a particular step in the Joining Protocol. This type is an instance of the `enum` type `Notification`. Furthermore, for the organization of the Joining Protocol the `IcapPolicy` class is responsible. Thus the `IcapPolicy` class implements a method

```
LocalPolicyResponse joiningNotification(IcapJoiningNotification)
```

that accepts a message sent by the Joining Protocol and processes it by invoking the corresponding static method from the `JoiningProtocol` class.

4.3.2 Scaling In

In the code the Scaling In module described in section 2.5.2 is referred to as *Leaving Protocol*. It is implemented in the class

```
cliseau.central.policy.scaling.LeavingProtocol
```

This class contains public, static methods for each step that has to be performed during the leaving of a node. Those steps are assumed to be called in the right order and before a scaling in request is processed the class is assumed to be initialized, i.e. the following method is assumed to be called before any other method is called:

```
LeavingProtocol.init(LocalPolicy).
```

For all communications related to the Leaving Protocol, the class

`cliseau.central.delegation.IcapLeavingNotification`

is used. This class contains several fields for the transported information as well as a `type` field relating the current message a concrete step in the Leaving Protocol. This type is an instance of the `enum` type `Notification`. Moreover, for the organization of the Leaving Protocol the `IcapPolicy` class is responsible. Thus the `IcapPolicy` class implements a method

`LocalPolicyResponse leavingNotification(IcapLeavingNotification)`

that accepts a message sent by the Leaving Protocol and processes it by invoking the corresponding method from the `LeavingProtocol` class.

4.4 Testing

4.4.1 JUnit Tests

The JUnit tests are contained in the package `test`. The class `PolicyTest` contains JUnit tests that check the behavior of the IcapPolicy class. They test the basic functionalities of components of the CliCap system.

`PolicyTest`

1. allowIrrelevantEvents
   Checks that all events irrelevant for the automaton defined in section 3.3 are allowed.
2. rejectTokenReuse
   Checks that reuse of tokens if rejected.
3. allowTokenUse
   Checks that a normal use of tokens is allowed i.e., tokens are accepted as long as they have been generated by the web shop but not been used to confirm a payment.

`EventFactoryTest`

This JUnit class provides one test for each critical event presented in 3.2.

4.4.2 Mapping Test

The mapping tests are implemented as unit tests (no JUnit tests) in the class

`test.MappingTest`.

It tests if critical events are correctly mapped to keys that can be used within the Chord identifier circle in order to determine the node responsible for a concrete event. The test results are printed to the standard error stream. Thus you can run the following command to see only the test results of these tests as output on the command line:

```
$ ./clicap build start 1 > /dev/null
```

4.4.3 Scaling Test

CliCap's scaling-in and scaling-out modules are tested step by step as follows:

1. Determine the end state of the system after the scaling process, i.e. which servers still remain running after all scaling operations have been successfully performed. As an example we choose servers with identifiers 2, 16 and 45.

2. Run `FingerTableGen` to generate the corresponding finger tables as well as successor and predecessor of each of these servers and stored them into a text file:

   ```
   $ java FingerTableGen 2 16 45 > expectedFT.txt
   ```

3. Navigate to the web interface of CliCap and perform any desired scaling operations such that the resulting network should contain exactly those chosen servers (the one with identifiers 2, 16 and 45 in the example).

4. Open directory `clicap/log`, open the corresponding log files. Now we can verify the correctness of the implemented scaling mechanism in this particular test case by comparing the logged information with the one generated by `FingerTableGen`.

Future work: Automatic test for scaling can be implemented based on this approach: the test automatically choses an end network state, the corresponding finger tables, successor and predecessor pointers; After that it performs scaling operations repeatedly until the chosen network state is reached; Finally the test verifies the logged information by comparing them with the correct one generated at the beginning.

Chapter 5

Usage

In this section we assume that you setup the CliCap *project following the instructions in the README file or run* CliCap *on the fully configured virtual machine provided by us.*

5.1 Startup

The CliCap project offers a script generator `scriptGen.java` that generates a startup script for the CliCap project with a specified set of nodes. The generated script is printed to the standard output stream. Thus, it might be desired to redirect this output into a file that can be used as startup script. The script generator offers two modes to run: without any parameters or with parameters that specify the nodes to start.

No parameters. If the script generator is run without any parameters, it generates a startup script for 10 nodes with the IDs 2, 7, 8, 29, 33, 37, 48, 51, 60, 63. Therefore, in order to create a startup script named `clicap` for the startup of those 10 nodes one can run the following command:

```
$ javac scriptGen.java && java scriptGen > clicap
```

IDs as parameters. If the script generator receives any parameters, those parameters are interpreted as the IDs of nodes for which a startup script shall be created. The generator will adapt those IDs such that they are sorted in an increasing order and less than 2^6. Furthermore, all duplicated IDs are merged into one ID and the smallest ID is considered as the major node. Thus, in order to create a startup script named `clicap` for the startup of the three nodes 2, 10, 20 where 2 is the Major Node one can run the following command:

```
$ javac scriptGen.java && java scriptGen 2 10 20 > clicap
```

In the following we assume that a startup script named `clicap` has been generated. The generated startup script offers three options:

1. `build`
   Builds the CliCap project from source and generates the *jar* file.
2. `start`
   Expects the CliCap project to be build and starts the set of nodes for which the script has been generated. The process IDs of the generated processes are stored in a temporal file.
3. `stop`
   Expects processes to be started by the this startup script and kills them. For this the process IDs from the temporal file written by the `start` option are used.

In order to build and start the CliCap project you can run

```
$ chmod +x clicap
$ ./clicap build start
```

After that the IDs of the started servers will be printed out. In order to stop those servers you can run

```
$ ./clicap stop
```

However, this will only stop the servers started by the last call of `./clicap start`, not the servers started by any previous call and not those started using the web interface. Those servers started via the web interface have to be removed via the web interface before any servers are stopped using `./clicap stop`. The reason is that the `stop` command of the generated script is only able to kill processes started by the corresponding `./clicap start`, any further process will have a different ID and therefore is unknown to the startup script.

Note: *SQUID* should be restarted every time we use the tool. You can run the following command after launching CliCap:

```
$ sudo service squid3 restart
```

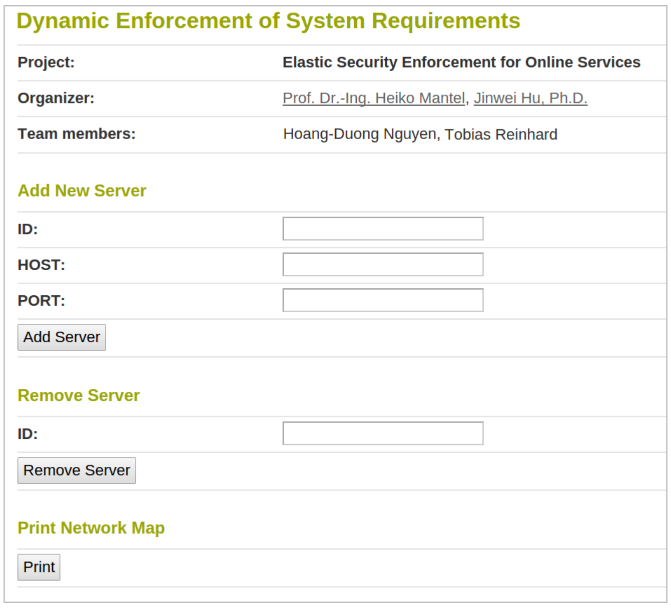
5.2 Web Interface

In order to visit the web interface presented in figure 5.1 you can navigate to:

`http://tomato.mais.tu-darmstadt.de/clicap`

Using this web interface it is possible to add and remove nodes from the current setup. In order to add a node the following fields must be filled in

1. `ID`: The ID of the node that shall be added. This ID is expected to be positive and less than 2^6.
2. `HOST`: The domain of the node that shall be added, for instance *localhost*.
3. `PORT`: The port of the node that shall be added. This port is expected to be freely usable by the any user without the need of any further right (like super user privileges.)



Dynamic Enforcement of System Requirements

Project:	Elastic Security Enforcement for Online Services
Organizer:	Prof. Dr.-Ing. Heiko Mantel, Jinwei Hu, Ph.D.
Team members:	Hoang-Duong Nguyen, Tobias Reinhard

Figure 5.1: Web interface for the management of the nodes.

The web interface expects the given values not to be in use by any other node. In particular, this is important to keep in mind when choosing the port as the port is not specified explicitly by the user when generating the startup script. There are two ways for the choice of the port. The first is to take a look in the startup script. The second and more convenient one is to choose a port bigger than

$$\max\left\{port_{last} + 3, \quad 10000 + 3 \cdot numNodes\right\}$$

where $port_{last}$ is the port of the last node added using the web interface or 0 if no such node exists and *numNodes* is the number of nodes started by the generated startup script. By this it is easy to see that it is a good idea to choose a large port number if it is not clear which ports are currently in use. Furthermore, it is also important to wait for the confirmation of the last adding or removing of a node before adding or removing another one.

Using the web interface it is also possible to remove a currently running node. For this it is not important if the node has been started using the startup script or using the web interface. The only restriction is that the node must not be the major node. To check this one can use the *Print Network Map* option. To remove a node just fill in the ID field and press *Remove Server*.

The *Print* option prints a graph plotting the current state of the whole network. The major node is illustrated as a blue server. The directed edges of the graph connect each server to all servers contained in its finger table. For instance figure 5.2 shows a setup with 15 nodes and node with ID 2 as major node.

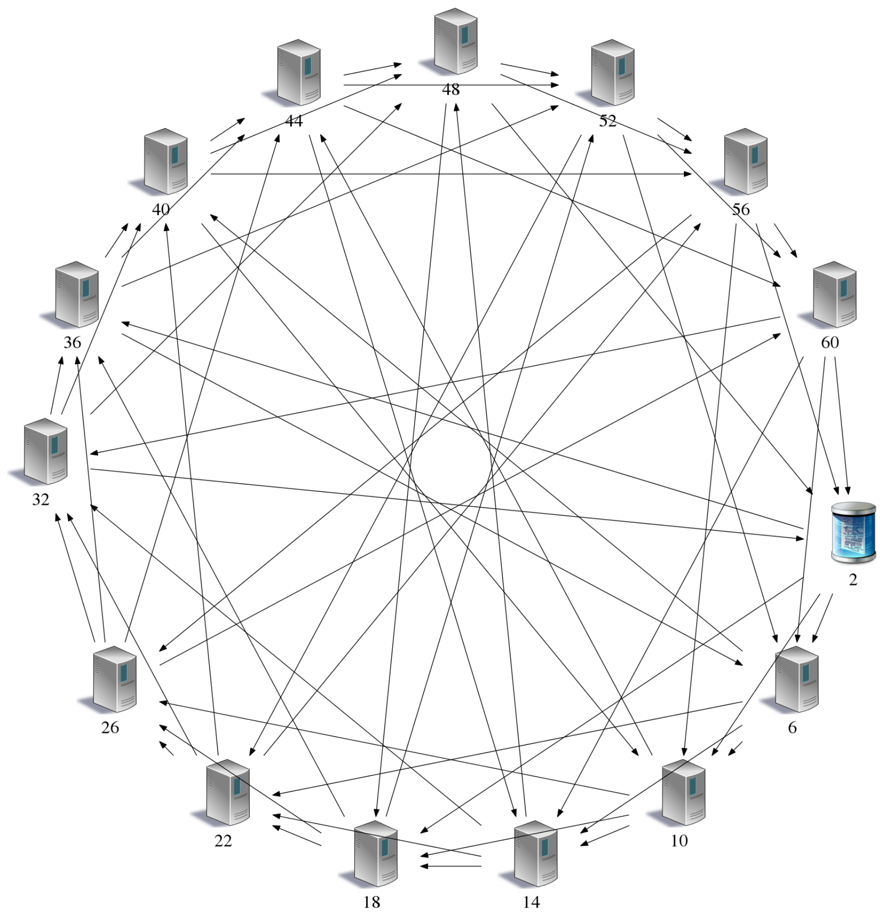


Figure 5.2: Network map of a system containing 15 servers.

Bibliography

[1] R. Gay, H. Mantel, and B. Sprick, "Service automata," in *Proceedings of the 8th International Conference on Formal Aspects of Security and Trust*, FAST'11, (Berlin, Heidelberg), pp. 148–163, Springer-Verlag, 2012.

[2] R. Gay, J. Hu, and H. Mantel, "Cliseau: Securing distributed java programs by cooperative dynamic enforcement," in *Information Systems Security* (A. Prakash and R. Shyamasundar, eds.), vol. 8880 of *Lecture Notes in Computer Science*, pp. 378–398, Springer International Publishing, 2014.

[3] "Tomatocart: An innovative shopping cart. http://www.tomatocart.com/."

[4] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications," in *NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA*, (San Diego, UNITED STATES), 02 2014.

[5] "Squid: Optimizing web delivery. http://www.squid-cache.org/."

[6] S. Götz, S. Rieche, and K. Wehrle, *Lecture Notes in Computer Science*, vol. 3485.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, (New York, NY, USA), pp. 149–160, ACM, 2001.

[8] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: enforcement mechanisms for run-time security policies," *International Journal of Information Security*, vol. 4, no. 1-2, pp. 2–16, 2005.

[9] "Greasyspoon: Icap server. http://greasyspoon.sourceforge.net/."