Modeling User Interactions for (Fun and) Profit:
Preventing Request Forgery Attacks on Web Applications

Karthick Jayaraman  Paul G. Talaga  Grzegorz Lewandowski
Syracuse University  Syracuse University  Syracuse University
kjyaram@syr.edu  pgtalaga@syr.edu  grlewand@syr.edu
Steve J. Chapin  Munawar Hafiz
Syracuse University  University of Illinois at Urbana-Champaign
chapin@syr.edu  mhafiz@illinois.edu

ABSTRACT
The goal of a web-request forgery attacker is to manipulate the intended workflow of a web application. Applications that fail to enforce the designer-intended interactions are vulnerable to this type of attack. This paper proposes a systematic methodology for designing web applications to strictly enforce the designer-intended interactions. Our approach captures workflow using the Web DFA model and applies four design patterns to strictly enforce the intended interactions. We argue that using patterns in conjunction with a Web DFA model produces web applications that are secure from request forgery attacks by construction; moreover, our mechanism could be useful in designing workflow-based applications in other domains.

1. INTRODUCTION
Web-request forgery attacks such as cross-site request forgery (CSRF) and workflow attacks can adversely affect the privacy and confidentiality of victim users. The root cause of these attacks is a weakness in the construction of web applications; most web applications are not designed to be strict enough to enforce the intended user-application interactions. Both variants of web request forgery attacks violate the intended interactions assumed by an application developer. In a CSRF attack, a malicious site forges and injects a request into a victim user’s active session with a trusted site. When Alice’s browser renders the malicious site hosts a crafted page that contains a request targeted to a trusted site (step 1). The trusted site stores a cookie for the ongoing sequence of valid transactions. In contrast to web attacks such as cross-site scripting and SQL injection, the malicious requests are well formed and valid with respect to the application. Hence, it is hard for a server to distinguish between a malicious and non-malicious request. Broadly, there are two types of web request forgery attacks: cross-site request forgery (CSRF) [11] and workflow attacks [6].

2. WEB REQUEST FORGERY ATTACK
In a web request forgery attack, an attacker manipulates an ongoing sequence of valid transactions. In contrast to web attacks such as cross-site scripting and SQL injection, the malicious requests are well formed and valid with respect to the application. Hence, it is hard for a server to distinguish between a malicious and non-malicious request. Broadly, there are two types of web request forgery attacks: cross-site request forgery (CSRF) [11] and workflow attacks [6].

2.1 Cross-site Request Forgery (CSRF) Attack
Figure 1 illustrates a typical CSRF attack. Alice visits a malicious site (step 3) while having an active session with a trusted site (step 1). The trusted site stores a cookie for the ongoing session in Alice’s browser (step 2). The malicious site hosts a crafted page that contains a request targeted to the trusted site. When Alice’s browser renders the malicious
Figure 1: Steps in a cross-site request forgery attack

- **Malicious site**
  - 3: Alice visits a Malicious site.
  - 4: Malicious site sends a crafted page.

- **Alice's site**
  - Step 3

- **Trusted site**
  - 1: Alice logs into a trusted site.
  - 2: Trusted site stores a session cookie in Alice's browser.
  - 3: Alice forwards request to trusted site with stored credentials from step 2.
  - 6: CSRF Exploit
    - Trusted site processes request from malicious site thinking Alice sent it

---

We illustrate our methodology with a running example: an online shopping cart implementation. We refer to it in each section.

### 3.1 The Web DFA Model

A web application's request behavior consists of a finite set of states which can be modeled using a DFA; we refer to this model as the Web DFA. In a Web DFA model, each state corresponds to a URL or specific functionality provided by parameters to a URL. In each state, the application delivers a web page to the user that could be used to issue subsequent requests. The transitions between states are HTTP requests. A web server hosting an application processes the request and produces the next page (goes to the next state in a Web DFA).

The Web DFA model for our running example, the shopping cart application, contains 10 states and 18 transitions (figure 3). The simplest interaction model of purchasing a single product is represented by transitions T1 through T10 (solid line transitions in figure 3). In this interaction model, a user comes to the home page, goes to sign in (T1), successfully completes authentication (T2), searches for products (T3), adds a product to cart (T4), continues to checkout (T5), confirms shipping and payment information (T6-T9), and completes order (T10). Transitions T11 through T18 represent some other valid user interactions. For simple illustration, we omit other valid interactions. For example, transitions T11–T15 illustrates that a user can at any point resume their shopping. Similarly, at any point the user may decide to go to the main page, or they might decide to logout and be moved to the main page. These transitions (originating at various states and ending at the main page state) are omitted. These omissions are for illustration purposes; they do not affect the final outcome.

---

### 3. METHODOLOGY FOR DESIGNING SECURE WEB APPLICATIONS

We propose a design methodology that drives the implementation to strictly enforce the intended user-application interaction in web applications. Our methodology has two steps. The first step is to model a web application's intended interactions using a Web DFA (section 3.1). The second step is to apply four design patterns to augment the DFA (section 3.2). Combining patterns with a Web DFA model produces a strict reference model that guides the implementation.

#### 3.1 The Web DFA Model

A web application's request behavior consists of a finite set of states which can be modeled using a DFA; we refer to this model as the Web DFA. In a Web DFA model, each state corresponds to a URL or specific functionality provided by parameters to a URL. In each state, the application delivers a web page to the user that could be used to issue subsequent requests. The transitions between states are HTTP requests. A web server hosting an application processes the request and produces the next page (goes to the next state in a Web DFA).

The Web DFA model for our running example, the shopping cart application, contains 10 states and 18 transitions (figure 3). The simplest interaction model of purchasing a single product is represented by transitions T1 through T10 (solid line transitions in figure 3). In this interaction model, a user comes to the home page, goes to sign in (T1), successfully completes authentication (T2), searches for products (T3), adds a product to cart (T4), continues to checkout (T5), confirms shipping and payment information (T6-T9), and completes order (T10). Transitions T11 through T18 represent some other valid user interactions. For simple illustration, we omit other valid interactions. For example, transitions T11–T15 illustrate that a user can at any point resume their shopping. Similarly, at any point the user may decide to go to the main page, or they might decide to logout and be moved to the main page. These transitions (originating at various states and ending at the main page state) are omitted. These omissions are for illustration purposes; they do not affect the final outcome.

#### Identifying Vulnerable Request Classes

After creating the initial Web DFA, the states in the DFA are classified into two categories. There are two types of states: non-
A state is **non-sensitive** if the transitions that lead to it do not have any side effects. An application does not modify the session data or database when processing these requests. A side effect free request does not modify an application’s state irrespective of the number of times that it is issued.

A state is **sensitive** if a transition that leads to it has side effects, i.e., it modifies application state. A transition with side effects could affect a user or the correctness of an application, if it is forged by an attacker. A web application needs higher guarantees to ensure that a user knowingly performed the transition and is not tricked into doing it. Transitions from non-sensitive to sensitive states and between two sensitive states should be protected from forgeries.

Figure 3 classifies each of the 10 states of the shopping cart application into appropriate categories. Of the 7 states performing the checkout transaction, from Shopping Cart Page through Order Confirmation Page, 4 are sensitive. The remaining 3 states present a form to a user whose information is then sent to the sensitive states. When a user decides to check out (T5), he/she first goes to the non-sensitive Shipping Info Page. In this state, the user is just presented a static web page to add shipping info; there is no change in

---

**Figure 2: A workflow violation in a purchase transaction where a user skips step 3**

**Figure 3: A Web DFA model for an online shopping cart application.**
the application state. Then the user provides shipping information (T6). The application takes the user to Submit Shipping Page. This state is sensitive since the application state is updated with the shipping information. The user, however, does not see a separate webpage. Instead, a script updates the application state and redirects (T7) the user to the Payment Info Page.

An HTTP redirect (T7 and T9) is used by the sensitive states to trigger loading the next state containing the next form in the workflow. In practice the same script could be used for data processing and form display based on the request type. We represent them as separate nodes in our DFA due to their different functions and security requirements.

### 3.2 Design Principles

We describe four design patterns for enhancing Web DFA models. The enriched Web DFA would act as a guide for developers to securely construct a web application. These patterns could be applied without the Web DFA model, but tying the design patterns with Web DFA makes the design process systematic, complete, and less cumbersome. In this section, we will describe each of the patterns and illustrate how the patterns are applied using our running example. Table 1 summarizes the patterns. The first three patterns protect a web application from CSRF attacks, while the fourth one protects from workflow attacks.

#### 3.2.1 Non-sensitive GET/ Sensitive POST

**Intent**

HTTP is the cornerstone of the World Wide Web. HTTP (version 1.1) defines eight request methods, each with its explicit recommended usage [7]. HTTP methods for reading and updating content follow the CRUD model of relation databases: PUT is used to create, GET to read content, POST to update content, and DELETE to delete content from a URL. GET and POST are more common in web applications, while PUT and DELETE are seldom used.

Despite the explicit specification of method roles, HTTP methods are often misused in web applications [1, 14]. For example, a developer who is considering whether to use HTTP GET, should follow these guidelines:

- GET should be used when a request does not affect the application state. The HTTP protocol defines GET as safe and idempotent: an HTTP GET request should not have any effect on an application’s state and the effect of multiple requests should be identical to that of a single request [7].

- For making sensitive requests, POST is favored over GET. It is harder for an attacker to forge a POST request, but GET requests are easily forged. This is because GET requests can be issued by putting URLs in the attribute header of many HTML tags (e.g., `img`, `iframe`, etc). When a user visits the page, GET requests are initiated without the user noticing them. On the other hand, forging a POST request requires either user interaction or JavaScript. To forge a POST request, an attacker has to coerce a user to submit a form. Alternatively, JavaScript programs can submit the form, but security setting in browsers prohibits untrusted JavaScript programs.

In practice, GET is often mistakenly used for making sensitive requests and modifying application state [4]: Bloglines sync API uses GET request to mark unread items as read, Flickr API previously used GET to delete a photo set, del.icio.us API uses GET to delete a post from the site, etc are some examples. Implementing a GET request in a web application is typically easier and results in less code than a POST request, possibly explaining their improper use.

Web application developers arbitrarily choose request methods, instead of considering which one is the most appropriate. The request type is treated as an implementation detail. Since the factors that influence the choice, such as whether the request is expected to have side effects or not, are known during design, it is best to determine the appropriate request type during design.

**Forces**

The following forces should be considered when choosing to use this pattern:

- Web applications should choose the most appropriate HTTP request type for each request.
- Choosing the wrong request type would facilitate request forgery.
- Choosing the most appropriate request type is best done during design.

**Solution**

Identify the type of processing and side effects associated with each request during the design phase and use this information to choose the appropriate HTTP request method.

Strictly use POST for any request of a sensitive state, i.e. it modifies database or a web application’s session data. Use GET for non-sensitive requests that do no have side effects.

There are certain requests that may have side effects, but they may still be considered side-effect free. For example, a request to visit the index page of a web site may automatically update page visit statistics. However, these statistics may not be considered as part of the application state. Therefore, such requests may still be considered non-sensitive and implemented as GET requests.

**Example**

This section describes how the pattern is applied to augment the Web DFA model in figure 3. Figure 4 shows the modified Web DFA.

Transitions to non-sensitive states are not expected to have any side effects; they can be implemented as HTTP GET requests. All transitions to the non-sensitive main page or the product search page can be implemented as HTTP GET requests. On the other hand, transitions to sensitive states should be implemented as POST requests.
Table 1: Design patterns to prevent web request forgery attacks

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-sensitive GET/ Sensitive POST</td>
<td>Choose the correct type for an HTTP request.</td>
</tr>
<tr>
<td>Secret-token Validation</td>
<td>Use a secret token whenever a sensitive request is made to distinguish between genuine and forged requests.</td>
</tr>
<tr>
<td>Intent Verification</td>
<td>Add an additional verification step to a request to verify whether a user intends to issue the request.</td>
</tr>
<tr>
<td>Guarded Workflow</td>
<td>Check preconditions and postconditions for each transition.</td>
</tr>
</tbody>
</table>

Figure 4: Web DFA for our shopping application with appropriate request type

**Consequences**
Applying the pattern has the following consequences.

- *Easy to understand.* Applying this pattern would make a web application easy to understand. Each request becomes intention-revealing; its type gives a hint of the operation to be invoked.
- *Weak Defense.* Attackers can forge POST requests [16]. This pattern provides the first layer of defense; more mechanisms are necessary following the defense in depth principle [17].
- *Increased complexity.* Applying this pattern could make a web application more complex. The simplest option for implementation is to use one HTTP request for all purposes; GET is the most suitable candidate. Having more than one HTTP request type would add more complexity; however, this is essential complexity [5] to make a web application secure.

**Known Uses**
phpBB and punBB are multi-user message board applications. osCommerce is an online shopping cart application. In all three web applications, HTTP GET requests are used only for non-sensitive requests, and POST is used otherwise.
3.2.2 Secret Token Validation

Intent
Strictly using POST to make sensitive requests provides a weak defense. An attacker can adjust the attacks to be form-based or use JavaScript to automatically invoke a request. Replacing GET with POST makes it harder for an attacker to launch an attack, but it is not hard enough to prevent the attack.

The underlying problem that enables cross-site request forgery is that a vulnerable web request can be repeatedly made. Typically, applications store session cookies in a web browser to customize each user’s request, but session cookies are attached whenever a browser makes a request. Session cookies are static; the same cookie is presented for all requests made from a user. Hence, an application has no way of distinguishing a legitimate request from a request that a user has unsuspectingly made on behalf of an attacker.

A user and a web application should have a secret that an attacker cannot know. If the secret is part of a web request, an attacker cannot forge it.

Forces
The following forces should be considered when choosing to use this pattern.

- HTTP requests can be repeatedly made.
- Session cookies are used to customize each user’s request, but they provide an insufficient mechanism to prevent forgery. This is because, for all requests to a domain, a browser automatically attaches that domain’s cookie.
- Cryptographic mechanisms could be used to create a unique token between an application and a user; an attacker cannot guess the token.

Solution
Use a secret token whenever a sensitive request is made. Protect the secret token, so that an attacker cannot know it. Verify each incoming request for a sensitive action to check that the secret token is present and correct.

In secret token validation, all HTML form tags that create HTTP requests include a random value as a hidden input field. This random value is passed to the server, and the server processes a request only after validating it. An attacker cannot access this random value since, 1) the value is available only in the web page given to the user, and 2) the security policy in web browsers prohibits the value to be shared.

Example
Consider the Web DFA model of the online shopping application in figure 4. All transitions to sensitive states are attractive targets for request forgeries. The processing of these requests should additionally incorporate a secret-token validation technique.

Consequences
Applying the pattern has the following consequences.

- Strong Defense. Secret tokens offer very strong protection with minimal computational overhead.
- Need for Protecting Secret. The session secret should be protected from attackers. One-time-use token values per form can be used, but they increase complexity and overhead.

Known Uses
This pattern is widely used for preventing cross-site request forgery attacks. phpBB, a message board application, adds a session identifier additionally as a hidden field to all web forms. The server-side scripts validate requests based on the session identifier. phpMyAdmin is a web application used for remotely administering MySql database. phpMyAdmin associates a random token for each session and adds the random token as a hidden field in forms.

3.2.3 Intent Verification

Intent
CSRF is a form of confused deputy attack [8]. The victim user, whose browser is making the request, does not know that he/she is being attacked. The user is tricked into submitting a request on behalf of the attacker. If a user is always asked before his/her browser sends a request, the user knows when he/she is about to be tricked by an attacker. Consequently, there will be no CSRF attacks. However, asking for consent at every step is impractical as users will find it annoying. There should be a lightweight approach that ensures usability of the application while assuring the integrity of each submitted request.

Many web applications use long expiration values for their browser cookies to keep a user continuously signed in. The cookies are used to track a session as well as to keep a user logged in so that users revisiting a site will not need to re-login. Applications which use long expiration values for their session cookies are highly vulnerable to CSRF.

Forces
The following forces should be considered when choosing to use this pattern.

- Users do not know when they are tricked by an attacker into a CSRF attack.
- Web applications should verify the intent of each submitted request.
- The intent verification reduces the usability of the application.

Solution
Introduce an additional verification step in the beginning of each transaction for verifying intent, i.e ascertaining that the correct user has consciously issued the request.

Force a user to re-authenticate at the start of a transaction.
Alternatively, use a CAPTCHA [18] to ensure that a request is coming from a consenting human.

Example
In a Web DFA model, the transitions from non-sensitive to sensitive states are the stepping stones for initiating a transaction. An attacker can host a CSRF attack on a non-sensitive page, but he/she is not gaining anything. It is when the attacker is stepping from a non-sensitive page to a sensitive page, is he/she starting a transaction. These stepping stones should be hardened using an additional intent verification step. The subsequent steps could proceed if the intent has been verified at the stepping stone.

In the Web DFA model (see figure 4) of our running example, the stepping stone to the checkout transaction is transition T4: adding a product to a cart. The web application should verify the intent of the user on this transition.

However, intent verification at T4 will hinder usability. A user, who is adding a lot of products to the shopping cart (following the T4-T11-T12-T4 cycle), has to verify for every product added (T4). Clearly, this is annoying. A better way is to check on transition T5 instead, when the products have been added to the cart and the user is opting for checkout. Real online shopping applications, such as www.amazon.com, verify a user at this step.

Consequences
Applying the pattern has the following consequences.

- **Informed User**: A victim user is informed when he is unsuspectingly initiating a sensitive request on behalf of an attacker.
- **Better detection of bots**: As a side effect of applying the pattern, web applications may distinguish Internet bots from real users.
- **Hindered Usability**: The verification step might be annoying for a user legitimately using the application.

Known Uses
Several web applications employing long login timeouts verify the user intent at the stepping stones of transactions. Both www.ebay.com and www.amazon.com allow users to search for products and add them to the shopping cart using long-term login. However, when a user tries to initiate a checkout transaction, the web application requests for a username and password. The checkout transaction is initiated only after correctly executing the verification step. If a subtask does not strictly check that its preconditions have been met, an attacker can violate the conditions and invoke the task nevertheless. Workflow attacks attempt to create an unintended interaction, in which certain subtasks are skipped by an attacker.

Forces
The following forces should be considered when choosing to use this pattern.
- Subtasks in a workflow should be executed in a pre-defined order.
- Attackers want to manipulate the normal execution order.
- Subtasks have preconditions that a caller should satisfy before invocation.

Solution
Identify the preconditions for each subtask in a workflow during design. During implementation, add checks to verify that all the preconditions are satisfied when a caller calls a subtask, otherwise identify it as a workflow violation.

Each of the subtasks have a set of preconditions. After invocation, each subtask creates a set of postconditions, which becomes the set of preconditions for the next subtask in the sequence. The precondition of any subtask is the union of postconditions of all the preceding subtasks. For each subtask, that should strictly follow a sequence of subtasks \{subtask_1, subtask_2, ..., subtask_{n-1}\},

\[
postconditions_1 \cup postconditions_2 \cup ... \cup postconditions_{n-1} \subseteq preconditions_n
\]

The design specification should outline an exception handling procedure for failing preconditions. The exception handler may either direct the caller to execute a preceding subtask or terminate the transaction.

Example
Consider the checkout transaction in the Web DFA model of figure 4. The transaction comprises of four steps: opting for check out (T5), submitting payment (T6), submitting shipping (T8) and confirming order (T10). Each transition has preconditions and postconditions (figure 5). The postconditions in each transition are chained so that they become the preconditions of the subsequent transition. As a result, there is no way an attacker can skip intermediate steps in the checkout transaction.

Exception handling procedures can also be described for workflow violations. For example, if the pre conditions associated with T8: Provide Payment Information are not satisfied when processing T10: Confirm Order, the application may direct the user to a web page to provide the payment information.

3.2.4 Guarded Workflow

Intent
A workflow is essentially one compound task composed of subtasks that have to be executed in a particular sequence. Each subtask expects its caller to meet some preconditions. In a web application, the preconditions are constraints on session variables or the application’s contents in a database.
Preconditions

User signed in
Shopping Cart not Empty
Total Cost > 0
Shipping Info not Empty

User signed in
Shopping Cart not Empty
Total Cost > 0
Shipping Info not Empty
Payment Info not Empty
Payment Authorization not Empty

Postconditions

Legend: Non-sensitive State Sensitive State Transition that omits some intermediate states

Figure 5: A checkout workflow annotated with required preconditions and postconditions.

Consequences
Applying the pattern has the following consequences.

- **Design by Contract.** Each of the preconditions and postconditions are determined carefully during design. The implementation that follows checks the conditions. Hence, the chance of a workflow violation is minimized.
- **Hard to Determine Preconditions.** In practice, determining the appropriate preconditions might not be straightforward. There might still be workflow vulnerabilities after applying this pattern. However, careful design nearly eliminates the vulnerability.

Known Uses

**Directed Session** pattern [13] uses a different approach. An application using Directed Session exposes a single URL. All webpages are accessed via this single URL. A server, using session data, determines which page be serve to the client. This dynamic approach, however, does not support the functionality of a back button in a browser. The **Guarded Workflow** pattern combined with the Web DFA is a more systematic way of exploring preconditions; it also supports the back button of a browser.

Design by contract [15] is a software engineering theory that describes formal contracts among software entities. A contract is the set of preconditions that a caller must guarantee before calling a module and the set of postconditions that would hold after the call. This pattern is essentially an application of design by contract for modeling workflow transactions in web applications.

3.3 Defense in Depth

The catalog of four design patterns is an ideal example of defense in depth [17]. First, the Web DFA is augmented by applying Non-sensitive GET/ Sensitive POST pattern. It determines sensitive states where POST requests should be used. But even POST requests could be forged. Therefore, **Secret Token Validation** mechanism is added with each POST request. Another line of defense is to keep a user informed about his/her actions. Hence, **Intent Verification** pattern is used to introduce verification mechanism in the transitions between Web DFA states where the user is stepping from a non-sensitive action to the start of a sensitive transaction. Finally, **Guarded Workflow** pattern is applied to the Web DFA to enforce design by contract [15]. Together, these patterns create multiple layers of defense that successfully prevent web request forgery attacks.

4. RELATED WORK

Current work has proposed several mitigation methods for web-request forgeries. The objective of mitigation methods is to detect the attacks at runtime and can be categorized into methods for detecting workflow violations and methods for detecting cross-site-request forgeries.

Swaddler uses an anomaly detection approach for detecting workflow violations [6]. Swaddler is a server-side method that uses probabilistic models for characterizing the attributes of internal session variables and for associating invariants with blocks of code for detecting workflow violations. The detection effectiveness is dependent on the accuracy of learning the invariants associated with blocks of code.

CSRF mitigation methods can be categorized into defensive coding methods, client-side methods [10], HTTP refer-
requests containing the correct secret token are processed by the web application, and malicious sites cannot access secret tokens in the trusted web page because of the access restrictions in the web browser.

RequestRodeo [10] is a client-side technique that avoids CSRF attacks by removing implicit authentication information, such as cookies and authorization fields in the header, from requests whose target URL and the URL of the web page from which the request originate do not conform to the same-origin policy. Kershbaum [12] proposes referer header validation. Barth et al. [3] describes the login CSRF attack and proposes strict referer header validation over HTTPS as a solution. NoForge [11] elegantly combines cookie-based and URL-based session management schemes to defend against CSRF attacks. It adds a secret-token to all the URL in the web page using a server-side proxy. Because only genuine requests would carry the secret-token and a malicious site cannot access it, forged requests are detected.

In contrast to mitigation methods, the objective of this paper is to present a systematic methodology for constructing web applications to avoid the attacks in the first place. Also, each mitigation method addresses only certain forms of request forgeries. On the other hand, the proposed methodology leads to applications that are secure from several forms of forgeries by design. Also, the cost of applying mitigation methods after the application has been built is costly compared to incorporating security in the application by design.

Prior work has used formal models similar to the Web DFA for testing and model-checking web applications [2,19]. Our work uses the Web DFA for creating a design methodology for web applications that are secure from request-forgery attacks.

5. CONCLUSIONS

We presented a novel method for designing web applications. We use a formal methodology, based on finite state automaton, in conjunction with design patterns to model and enforce intended user-application interactions in web applications. In the future, we plan to build tools to allow designers to build and analyze Web DFA models.

Web applications are heavily used and attractive targets of exploitation. Both patching applications and providing design fixes after deployment has proved to be prohibitively costly. Therefore, we need better methodologies for building web applications that are secure by construction.

Acknowledgments

We thank Dr. Jim Fawcett and the attendees of PLoP ’09 for their suggestions and comments.

6. REFERENCES

[16] Robert Auger. The Cross-Site Request Forgery (CSRF/XSRF) FAQ.