Position Paper: Why Are There So Many Vulnerabilities in Web Applications*?

Wenliang Du, Karthick Jayaraman, Xi Tan, Tongbo Luo, and Steve Chapin
Dept. of Electrical Engineering & Computer Science, Syracuse University
Syracuse, New York, USA

ABSTRACT

As the Web has become more and more ubiquitous, the number of attacks on web applications have increased substantially. According to a recent report, over 80 percent of web applications have had at least one serious vulnerability. This percentage is alarmingly higher than traditional applications. Something must be fundamentally wrong in the web infrastructure.

Based on our research, we have formulated the following position: when choosing the stateless framework for the Web, we ignored a number of security properties that are essential to applications. As a result, the Trusted Computing Base (TCB) of the Web has significant weaknesses. To build secure stateful applications on top of a weakened TCB, developers have to implement extra protection logic in their web applications, making development difficult and error prone, and thereby causing a number of security problems in web applications. In this paper, we will present evidence, justification, and in-depth analysis to support this position.

1. INTRODUCTION

In recent years, we have observed a proliferation of vulnerabilities in the web. Although the Web has only become widespread within the last ten years, the number of some particular vulnerabilities on the web has already exceeded those in traditional applications. According to a recent report [34], over 80 percent of websites *have had* at least one serious vulnerability, and the average number of serious vulnerabilities per website is 16.7.

The phenomenon of such a proliferation of vulnerabilities is hard to understand. Many may attribute this to the number of applications in the Web. That can explain why web vulnerabilities outnumber traditional vulnerabilities, but it cannot explain why such a high percentage of web applications are vulnerable. This percentage is significantly higher than that of traditional applications. Moreover, because of the similarity to traditional client/server computing (browsers are the client, and web servers and application programs—such as PHP programs—are the server), we might expect to see vulnerabilities that mimic those in traditional applications, both in proportion and nature. Instead, we are seeing unique types of vulnerabilities that are quite rare, or non-existent, in traditional client/server applications. Because of the high percentage of vulnerable web applications and the sheer number of web applications, these flaws quickly becomes the top-ranked vulnerabilities facing systems today.

If web applications are like traditional client/server applications, what has caused those unique vulnerabilities, and in such a high quantity? There must be some fundamental difference between these two types of applications, and that may be the root cause of these unique vulnerabilities. This question has been pondered by many researchers, as evidenced by the published work in the literature. Although the existing work offers partial answers , the question has not been fully answered at the fundamental level. In this paper, we would like to make such an attempt. Our goal is to find the fundamental causes of the unique vulnerabilities in web applications.

Over the course of this pursuit, we have identified many causes, because when you change an angle or look at the problem at a different level, you see a different cause. Providing a laundry list of all possible causes is unlikely to provide useful insights to the problem. The cause must be fundamental, meaning the cause should be found among the points of convergence of many vulnerabilities in web applications. We are not necessarily looking for a single point of convergence; there may be several of them.

Based on our research, we have identified one convergence; many vulnerabilities in web application can be traced to this point. This is the stateless nature of the Web, i.e., many security problems found in web applications are caused by building stateful applications on this stateless infrastructure. Although it is well-known from the literature that the stateless nature of the Web has caused many problems, there has been a lack of in-depth study to answer why statelessness causes so many problems. This position paper answers such a fundamental question. In the rest of this section, we first review the stateless feature of the Web, and then give a brief summary of our main positions.

The Stateless Nature & Sessions. Web servers are designed to be stateless, namely, each HTTP request is processed by an independent server-side process or thread, even

*This work was supported by Award No. 1017771 from the US National Science Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

To Appear in The New Security Paradigms Workshop (NSPW), Marin County, CA, USA. September 12-15, 2011.

if two requests are related. This is in stark contrast to traditional client/server applications, which are mostly stateful. In stateful applications, the same server-side process/thread will be dedicated to a client, until the client terminates (e.g. `telnet`, `ftp`, and `ssh`). The main reason for the Web's stateless property is performance. Web servers usually serve a much larger client base than do traditional client/server applications, so they do not want any process to be tied up by a single client.

However, most web applications are stateful. A client's HTTP requests do indeed exhibit dependencies; this dependency relationship must be recognized by the server. For example, in a shopping-cart application, the products picked by a user need to be remembered when the user traverses from one page to another. This demands support for statefulness at the server, which is stateless by nature. *Sessions* allow stateful web applications to run on stateless infrastructure. When a user browses a web site, the server provides the user with a session ID, which is stored in the user's browser as a cookie. When the user sends other HTTP requests to the same server (within the lifetime of the session), the session cookie will be attached; therefore, the server can recognize these requests as being related (i.e., tying them to the same session), and thus provide support for statefulness.

To carry session-related data from one request to another request of the same session, web servers store these data in non-volatile memory, such as files and databases. After identifying the session of a request, the web server retrieves all the session data from these repositories. Session information, including session ID and session data, has a typical lifespan of minutes to hours, and expires after certain goals are achieved.

Our Discoveries and Positions. It seems that sessions have "emulated" the need for statefulness quite well from the functionality perspective, because they do a good job in preserving relevant state information. The question is whether this is sufficient for other purposes. To answer this question, we need to look at this emulation from another angle, the non-functional angle. If such an emulation is sufficient, it should also emulate the non-functional properties that are intrinsic to stateful programs. Unfortunately, this is not true.

There are four security properties that sessions do not emulate: preservation of trust state, data integrity, code integrity, and session integrity. These four properties are part of the Trusted Computing Base (TCB) in a stateful framework. Therefore, choosing a stateless framework without preserving these security properties, the Web has chosen a TCB that is severely weakened. An immediate consequence is that preserving those security properties now falls upon developers' shoulders. Namely, web application developers have to implement application-specific logic to maintain those security properties. It was mistakes in, or a lack of, the developers' implementations of these security properties that has caused the abnormally large number of vulnerabilities on the Web. In response, most existing work has focused on developing attack-specific methods to prevent those mistakes; we believe that the right approach should focus on enhancing TCB, providing systematic support for those security properties.

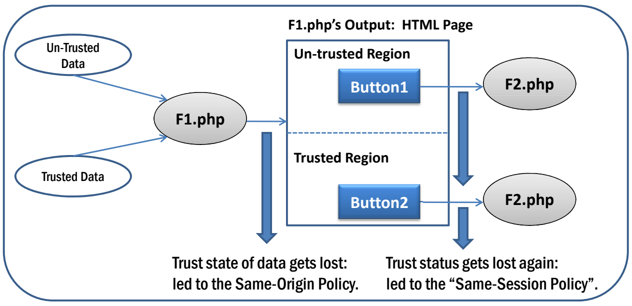

F1.php's Output: HTML Page
Un-Trusted Data
Trusted Data
F1.php
Un-trusted Region
Button1
F2.php
Trusted Region
Button2
F2.php
Trust state of data gets lost: led to the Same-Origin Policy.
Trust status gets lost again: led to the "Same-Session Policy".

Figure 1: Loss of trust state in web applications

Organization of the Paper. In Sections 2, 3, 4, and 5, we explain in great detail why the stateless nature of the Web has led to failure to maintain the aforementioned security properties, what impact this failure has caused, and how web applications have struggled to battle with this failure. In Section 6, we summarize our discoveries, which lead to the main position of this paper. We then present our position on what we should do to improve the situation.

2. PRESERVING TRUST STATE

The first security property that we will look at is the *trust state*, the trustworthiness of the data on which an application operates. In a typical stateful application, the application can, if it chooses to, track and make access control decisions based on such a trust state. Unfortunately, preserving trust states becomes impossible in the current Web infrastructure. We will show how trust state information gets lost within a single session. For the sake of simplicity, we use an example in our discussion. The example is depicted in Figure 1.

Initially, server-side programs (e.g. `F1.php` in Figure 1), know how trustworthy their data are, because they know where the data come from. However, when `F1.php` puts all the data together into one HTML page, and presents it to the client-side browsers, there is no systematic mechanism for the server to convey the data's trust state to browsers, so the trust state gets lost in the process.

Even if the above problem is fixed [1], when control returns to the server, the trust state information gets lost again. Program control returns to the server when the browser invokes a server-side script. For example, in Figure 1, `F2.php` is invoked when a user clicks a submit button on the web page. Many things can trigger such an invocation. The figure only shows one scenario; other scenarios include clicking a URL, event-triggered invocation, invocation directly from JavaScript code, etc. When the control returns back to the server, the server can only tell whether the invocation belongs to the same session or not, not whether the invocation is triggered by trusted or untrusted contents.

2.1 Impact on Security & Access Control

Failure to preserve trust states means that access control in web applications cannot be based on accurate and fine-grained trust states. This has an immediate impact on security. On the client side, the inability to receive the trust state from the server forces browsers to use a much more coarse granularity–origin–in its access control, trusting all data from the same origin equally and giving them the same

[1] One of our earlier papers fixed this problem [11].

privileges. This is the so-called same-origin policy (SOP). On the server side, the inability to discern the trust state of invocations means servers can only use sessions as the finest granularity in their access control, giving all invocations from the same session the same privileges. We refer to this as the *same-session policy (SSP)*. The inadequacy of these access control models has been pointed out by various studies [1, 4, 5, 10, 16, 21, 23, 27].

Such coarse granularity may have been sufficient for the nascent Web ten to fifteen years ago, when a typical HTML page's contents were usually homogeneous, i.e., they came from the same–usually trusted–resource. However, through the evolution of the last decade, the landscape of the Web has changed significantly. Nowadays, contents in web applications come from diverse sources. We describe three representative scenarios:

Untrusted contents: Many web pages contain user-provided content such as blogs, comments, feedbacks, user profiles, etc. These contents are less trustworthy than the content generated by the server. Servers know this fact, but due to the lack of a trust-preservation mechanism, once a server integrates these diverse contents into a single web page, all contents share the same privileges, regardless of the trustworthiness of the sources. If these contents are just passive data, we will not see much of a problem. However, the contents can also be active contents, such as Javascript code or action-invoking HTML tags; if these actions are malicious, we have a problem that can lead to a cross-site scripting (XSS) attack, the No. 2 ranked attack on the OWASP Top Ten list [26]. Since the active contents are executed on the browser side, if browsers do not know how trustworthy the contents are, they cannot conduct appropriate access control on the actions.

Client-side extension: Many web applications allow client-side extension, i.e., they include links to third-party programs in their pages, and run those programs in the browser. A common example is web advertisements, which usually contain code (JavaScript, Flash, etc) from ad networks. Facebook applications are another example of client-side extension. In Facebook, third-party applications can be embedded into a user's Facebook page; these applications contain code from a third-party application server. iGoogle's gadgets are another examples of client-side extension.

Servers know that third-party content is not as trustworthy as that generated by the server itself, but unfortunately, there is no easy way for servers to tell browsers that these contents as untrustworthy; the trust state of the contents is not preserved once the data leave the server.

In this situation, web applications have to assume that the third party will not provide them with malicious contents. Such an assumption is very fragile. For example, in a recent event (September 2009), an unknown person or group, posing as an advertiser, snuck a rogue advertisement into the New York Times' pages, and successfully compromised the integrity of the publisher's web application using a malicious JavaScript program [31].

Server-side extension: Some web applications include server-side code that are developed by a third party. For example, `Elgg` is an open-source social network application. It was designed as an open framework, allowing others to extend its functionality. There are already hundreds of third-party extensions. To use these extensions, administrators of `Elgg` need to install them on the server side. Once these extensions are installed, they have the same privileges as the native code of `Elgg`.

There are two problems in this architecture: first, there is no mechanism for the web server to label how trustworthy an extension is; second, even if we could perform such a labeling, the stateless nature of the Web does not allow us to preserve such a trust state throughout the lifetime of a session. As a result, the actions initiated by extensions share the same privileges as the native server-side code.

Although no malicious extensions have been reported so far for `Elgg`, vulnerable extensions have been discovered. When a vulnerable server-side extension is installed on the server, any security problem (such as an XSS problems) will affect the entire web application because of the privileges they receive. Although `Elgg` has spent a lot of effort securing its own core code, they do not have an effective method to contain the security breaches caused by vulnerable extensions.

2.2 Further Impact: on Access Control Model

The inability to differentiate trustworthiness levels of data and code has led to the incorrect design of the Web's access control models (i.e. the same-origin and same-session policies), which clearly violate some of the vetted design principles summarized by Saltzer and Schroeder [29]:

1. **Separation of Privilege:** If possible, privileges in a system should be divided into less powerful privileges, such that no single accident or breach of trust is sufficient to compromise the protected information.

2. **Least Privilege:** The protection model should be able to limit the interactions of principals based on their trustworthiness. Essentially, a principal should not have more privileges than required for its legitimate purpose.

Browsers and servers fail to follow either of the above principles. This is not because of the designers' ignorance of the trust state of the action initiators; instead, this is caused by the web infrastructure' failure to preserve trust states. We believe that the current access control models of the Web infrastructure are fundamentally inadequate to address the protection needs of today's Web; they need to be redesigned. In the new design, the trustworthiness of data needs to be used as one of the bases for access control. This allows us to provide a finer granularity in access control. A well-designed access control model can simplify application developers' tasks by enforcing much of the access control within the model, freeing developers from such a complicated and error-prone task.

2.3 Extra Efforts for Preserving Trust State

Because of the inadequate access control on the browser side, when including third-party programs in a web page, web applications have to ensure that these programs are not malicious; otherwise, once reaching browsers, these third-party programs will have the same privileges as those first-party programs, putting web applications in great danger. To reduce risk, applications have to verify those programs before integrating them into web pages. Facebook took this approach before including a third-party application into its site. Advertising networks also conduct code verification on JavaScript advertisement programs before accepting them [8]. These verification strategies are expensive to use and are not fool-proof: numerous security breaches have been reported [31].

Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) attacks are the inevitable consequence of the Web's inappropriate access control models. Existing work has proposed solutions to protect against these attacks. Approaches to XSS include taint-tracking [9,24,28], pure client-side solutions [19,33], pure server-side approaches [3], and co-operating defenses [13]. Similarly, cross-site-request forgery solutions can be categorized into client-side methods [14], HTTP referrer header validation [17], proposals for new headers [2], and secret-token validation techniques [15]. All these solutions are attack-specific patches to the application, framework, or browser. None of them tries to fix the root cause, i.e., the failure of the web infrastructure to preserve trust states and the resulting inappropriate design of the access control model.

3. PRESERVING CODE INTEGRITY

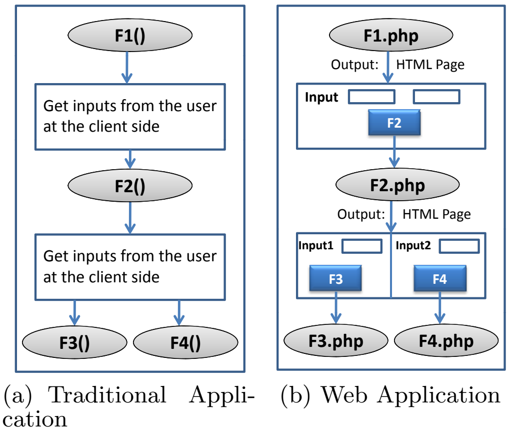
Being able to preserve code integrity is essential for security. In this section, we compare the stateless web infrastructure with the traditional stateful framework, in terms of how they preserve code integrity.

In the traditional client/server framework, servers use both synchronous and asynchronous modes to to get clients' input. In synchronous mode, the server stops at a point to wait for input from the client. In asynchronous mode, the server does not wait for input; instead, when input comes, a callback function on the server side is invoked to process the input. These two different modes are used for different scenarios.

Synchronous mode is used when functions can only be triggered if certain conditions are met or states are reached. For example, if the server needs to get the client's credit card information before going to the next step—sending a mail order to the warehouse—the server needs to wait for the credit card information, i.e. synchronize with the client at the waiting point. Asynchronous mode is mostly used in scenarios when a function invocation does not depend on a pre-condition; as long as the inputs come, the function can be triggered. Synchronous mode can only be used by stateful applications, because of the need for client-server synchronization, while asynchronous mode can be used by both stateful and stateless applications.

Web applications, because of their stateless nature, can only use asynchronous mode, regardless of whether synchronization is needed or not. When a web server receives an HTTP request, it spawns a process (or gets an idle process from the pool) for this request. For example, in Figure 2(b), `F1.php` is executed in a process, and during execution, the process will not request any user input. If user input is indeed needed, the process will create an HTML page, send it to the client's browser, and then terminate; it does not wait for the return of the client's input. When the client returns his/her input, a new process will be triggered to process the input. This is a purely asynchronous mode.

In asynchronous mode, the server needs to register a callback function, such that when input is returned, the function can be triggered to process the input. In web applications, callback functions, in the form of the names of server-side programs (e.g. `F2.php` in Figure 2(b)), are embedded in the HTML page sent to the client. The function names are usually put in the `action` field of a form as a URL. After a client fills out the form, and clicks the `submit` button, the callback function will be invoked on the server side. Figure 2(b) gives an example. In the example, `F2.php` registers two callback functions, `F3.php` and `F4.php` in its generated HTML page. Clients can choose which one to invoke, depending on which button they click.



(a) Traditional Application (b) Web Application

Figure 2: Difference in Control Flow

3.1 Impact on Security

In terms of security, the major difference between these two modes is how convenient it is to maintain the code integrity: using synchronous mode, it is much easier to preserve code integrity than using asynchronous mode. Using synchronous mode, server-side programs can encode their state transitions (i.e. control flow) within their code (see Figure 2(a)); the integrity of the control flow is protected by the server. Although clients can affect the control flow via their input, they cannot directly change the control flow of the server program without changing the server-side code (difficult for attacks).

On the contrary, in asynchronous mode, the invocation of the callback function—one form of state transition—is decided by clients, who can decide which callback function to invoke, when to invoke them, and in what order. For instance, in the example illustrated in Figure 2(b), although the server intends the client to trigger a particular function (e.g. F2), and only embeds that function in the first HTML page, there is no guarantee this will, in fact, occur: the client can instead trigger F3, bypassing F2. All he/she needs is to know the name of F3 (not a secret in web applications) and construct an HTTP request with F3 being the target URL.

Many web applications are better implemented using the synchronous mode, if possible, because ensuring the integrity of the state transitions is essential. Unfortunately, the stateless web infrastructure only provides asynchronous operation, making the development very unnatural. Let us look at a typical web application example. In the online shopping web application, a typical flow of control has three steps: the customer selects products (G1), he/she then fills out the payment information (G2), and after verifying the payment, the server sends an order to its warehouse, requesting the selected products to be mailed to the customer (G3). The control sequence is G1 → G2 → G3. Using synchronous mode, this control flow can be easily enforced by invoking the functions in a sequence, ensuring that the advancing to the next step is only possible if the previous step is finished. Such an enforcement is automatically (i.e. implicitly) carried out as long as the execution order is specified in the code.

Unfortunately, enforcing such sequences using asynchronous mode is quite hard, because the client controls the order of callbacks to the server. For example, in Figure 2(b), the order of execution is indeed specified by the programmers (F2 first, and then F3 or F4), but there is no automatic mechanism to prevent the client from invoking F3 first, skipping F2.

3.2 Extra Efforts to Achieve Code Integrity

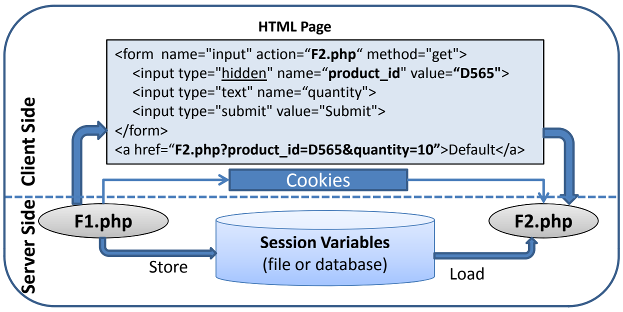
To preserve the integrity of control flow, web applications have to resort to extra help: this is done either by extra program logic in web applications or by using specialized frameworks. Developers commonly use two methods to enforce control flow, namely *interface hiding* and *validation* [12]. In the case of interface hiding, the basic idea is to control the display of URLs in the web page. Whenever the application prepares a web page for the user, it only displays the URLs that the user is entitled to access. As long as the user issues requests by interacting with web pages, this method will enforce control flow. The directed session design pattern [18] describes the same idea. In the case of validation, the server side of the application accurately models the state of the client side of the application using session variables. The values of the session variables determine what further request the application can process. For example, most web applications require users to be authenticated first, before being able to use services. These web applications usually use a variable to record whether the user has logged in or not; if not, users will be directed to the login page. This is how they enforce the simple "login → service" sequence.

Construction frameworks such as the Spring framework [6] provide support for separating page navigation from business logic. However, the enforcement of control flow should still be done inside server-side scripts using validation methods. *Bayawak* [12] is a server-side method that transparently enforces control-flow integrity in the application based on a security policy that describes valid request sequences. The basic idea is to diversify the application for each session, making it infeasible for attackers to predict how to form a valid request sequence. *Ripley* [32] is a server-side method for verifying the correctness of client-side behaviors in web applications. This is done by running a redundant copy of the client on the server, and comparing their behaviors.

4. PRESERVING DATA INTEGRITY

Being able to preserve data integrity is essential for security. In this section, we compare the stateless web infrastructure with the traditional stateful framework, in terms of how they preserve data integrity.

Just like any traditional applications, web applications have two types of data: global data and local data. Global data are usually stored in global variables, and they are accessible within the session by code at all scopes. Local data are usually stored in local variables, and they can only be accessible within a particular scope, e.g., within a function. During execution, global variables are usually stored in the heap area, while local variables are stored in the stack area. As long as the process is still alive, the data are still there, and accessible. For applications running in a stateful framework, a process usually lasts throughout the entire session, so data are always available. Unfortunately, for web applications, there is a problem: requests, even if they belong to the same session, are processed by different processes, making passing data via internal memory infeasible. To make data available for subsequent requests, web applications use two primary methods to store data for later access:



```
<form  name="input" action="F2.php" method="get">
    <input type="hidden" name="product_id" value="D565">
    <input type="text" name="quantity">
    <input type="submit" value="Submit">
</form>
<a href="F2.php?product_id=D565&quantity=10">Default</a>
```

Figure 3: Data Flow in Web Applications

Through files or database. The first process can save data in a file or database. When the second process starts, it can fetch the data from there. To make this a transparent process, PHP introduces the *session variables* concept. The first process stores the data in session variables, which will be dumped into a file or database (depending on the configuration) before the process exits; the next process can automatically reconstruct those variables from the file or database, when it calls `session_start()`. An illustration is given in Figure 3.

Through browsers. The first process can embed data in its generated HTML page, making them available for the subsequent requests from this page (see Figure 3). There are several ways to embed data. One is to embed data into hidden fields of a form; when users submit the form, the values in the hidden fields will be automatically attached to the HTTP request. A second method is to embed data directly in the URL of the target function, so when the invocation is sent to the target function through HTTP requests, the data is appended in the request. Another method is to send data to browsers as cookies, which will be attached to all subsequent HTTP requests.

The above two approaches are actually common in traditional stateful programs. The first method is like passing data via global variables, because session variables are basically global variables in web applications. The second method is like passing data via arguments. The hidden fields and data appended to URLs in HTML pages are very much like the arguments passed to the target function in the traditional programs.

4.1 Impact on Security

The second method (passing data through browsers) clearly has a security risk for web applications. The data passed between functions are now exposed to the clients, who can make arbitrary changes to the data. For example, in the example shown in Figure 3, both the `product_id` and `price` values will be returned back to the server as an HTTP GET request when the clients click the `submit` button. Unfortunately, the clients can easily modify the values of these two parameters before the HTTP request is sent from their browsers; or they can write a program to craft an HTTP request with any values they like. This is impossible in traditional programs, as data passed between functions are stored in the internal server memory, not exposed to users. Many web developers do not realize this fundamental difference between passing data in web applications and that in traditional applications, they end up introducing vulnerabilities in their applications. The A4 vulnerability, No. 4 ranked in the OWASP Top Ten list, is mainly caused by this type of mistake [26].

4.2 Extra Efforts to Achieve Data Integrity

There are two primary solutions to counter the aforementioned security risks. The first method is to use integrity verification, which can be done by attaching a MAC (Message Authentication Code) to the data, making data tampering easily detectable. The current web infrastructure does not provide such a mechanism. Until this systematic support is integrated in the web infrastructure, developers have to write their own program logic to secure data.

The other solution is to avoid passing data through browsers; instead, one can use global variables (i.e. session variables) to pass data between functions. This way, data always stay at the protected server side, never exposed to the clients. This method is indeed secure; however, by doing so, we are against a common software engineering principle, i.e., minimizing the use of global variables. Although the security problem is fixed, we pay the price by increasing the complexity of our programs, an inevitable consequence of using more global variables.

There is another problem if we use global variables to replace local variables. If a function can be called multiple times simultaneously (e.g. due to recursion or parallelism), the use of global variables makes it non-reentrant. In traditional applications, stacks, which are used for passing data, are good at handling the above situations, because each invocation of a function is going to get its own stack frame (and thus its own copy of the arguments), while using global variables, there is only one copy. Recursion may not be common in web applications, at least so far, but parallelism is inherent in the nature of the Web: users may simultaneously send multiple HTTP requests within the same session. To handle such a condition, the complexity of web applications will be inevitably increased.

5. PRESERVING SESSION INTEGRITY

In this section, we look at the design of the session mechanism itself, and compare sessions in both the web framework and the traditional stateful client/server framework. The differences between the sessions in these two frameworks, although subtle in behavior, are quite significant in security.

In the stateful client/server framework, a typical session usually comprises a client-side process, a server-side process, and a TCP connection between them (see Figure 5(a)). The TCP connection is identified by the source/destination IP addresses and port numbers, as well as by the TCP sequence numbers. When a server receives data from the client at different times, the data are considered as belonging to the same session if they come from the same TCP connection, i.e., the IPs and port numbers match accordingly, and the TCP sequence numbers fall into the current TCP window. We call the collection of these pieces of information the session identifier.

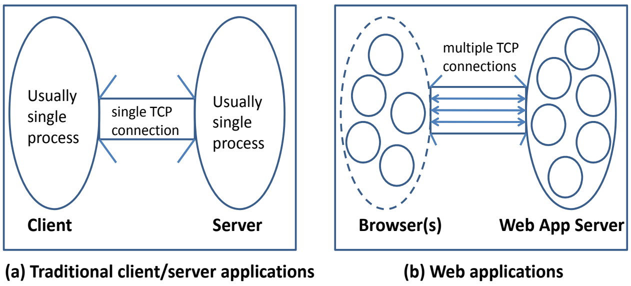


Figure 4: Similarity & Difference in Session Concept

In the web framework, a session is not bound to a single process on the client or server sides, nor is it bound to a single TCP connection; instead, a session is only bound to a unique number called a *session ID*. Requests in the same session can come from different client-side processes or different computers, through different TCP connections, served by different server-side processes, as long as they carry the same session ID (see 5(b)). In other words, the only session identifier provided by the underlying web infrastructure is session ID.

From the behavior perspective, what is achieved by the session ID in the web framework is similar to that in the stateful client/server framework (see Figure 5). However, the ways sessions are implemented in these two different frameworks cause significant differences in security. We analyze the differences in depth in this section, organizing the discussions based on the attacks that are related to sessions.

5.1 Session Hijacking

Session hijacking is an attack in which an attacker successfully injects data into an existing session between the victim and the server. To fool the server into treating the injected data as belonging to the victim's ongoing session, the attacker needs to use the correct session identifiers. For the stateful client/server framework, the attacker needs to discover the source/destination IP addresses, the source/destination port numbers, and the TCP sequence number. For the web framework, the attacker needs only to steal the victim's session ID. Although what attackers need to do looks similar in both frameworks, there are significant differences, mostly in how difficult it is to launch the attack.

Binding or Non-binding. In the stateful client/server framework, sessions are bound to two processes on two computers (one at the client side and the other at the server side); only these two processes can legitimately use the session. This is not true for web applications, in which only one end (the server end) is bound to a session; the client end is not. Therefore, the client can legitimately use a session from a different process or machine.

Because of the nature of binding in the stateful framework, session hijackers must spoof packets after knowing the session identifier. In particularly, they must spoof the client's IP address, TCP port number, and the TCP sequence number. For web applications, however, spoofing is not needed: once the attackers obtain the session ID, they can use the session like the session's legitimate user, from another computer.

One-way or Two-way. In TCP session hijackings, the response to the spoofed packets will still go to the spoofed client, not to the attacker, because of the spoofing. In other words, the attackers cannot see the response (unless they can observe the traffic between the legitimate client and server). We call this type of session hijacking one-way; by analogy, a two-way hijacking allows the attacker to see traffic both to and from the server. Limiting to one way increases the difficulty for session hijacking. In web applications, because no spoofing is needed, the reply will go to the attacker, making two-way session hijackings much easier.

Disclosure of Session Identifiers. It is easier for web applications to inadvertently disclose their session identifiers than the stateful client/server applications. In the latter case, the source port number and the TCP sequence number are usually transparent to programs: rarely are they referenced by programs (there is probably no API to obtain the sequence number). In web applications, it is easy to get the session IDs within the program (e.g. in PHP, `session_id()` returns the session ID). When something is so easily accessible, developers will soon make "creative" use of it, leading to potential disclosure of the session ID. For example, to allow users to share their private data (e.g. pictures) with friends, some web applications send the friends a URL of the data, with the session ID attached. This basically allows the friends, or any attacker able to read the message carrying the URL, to hijack the entire session.

5.2 Session Fixation

If stealing session identifiers is too difficult, attackers can try a different approach to get the session identifiers: setting or resetting victims' sessions using a known identifier. This type of attack is called *session fixation*. Once the attackers have successfully fixed the victim's session, they have all the information to hijack this session. Session fixation was rarely, if ever, heard of for traditional client/server applications. This is because parts of the session identifiers are decided by operating systems: when a victim process establishes a session with a server, it cannot set its source port, source IP, or the TCP sequence number, unless the process is a root process and wants to do packet spoofing. Therefore, it is impossible for a non-root malicious process (or a malicious machine) to "convince" the victim process to use a provided session identifier.

Session fixation is a common attack in web applications. Along with several other session-related vulnerabilities, they take the No. 3 position on the OWASP's Top Ten list [26]. The main reason is that session identifiers can be set by normal users, which opens a door for attackers. There are several mechanisms for setting the victims' session identifiers.

Through URL. In PHP programs, if users provides an SID parameter in their URL, `session_start()` will use this value as the session ID when creating a new session for the current HTTP request. For example, a rogue URL can look like this: `http://targeted_server.com/logon.php?SID=12345`. To get the victim to use this fixed ID, the attackers need to send this URL to the victims, and get them to click on this URL. After the victims enter their credential to `targeted_server.com`, they will get an established session with the server. Unfortunately, the session identifier, provided by the attackers, is not a secret anymore.

Through Cookies. In PHP programs, if users provide a cookie named SID in their HTTP request, `session_start()` will use this value as the session identifier when creating a new session for the current HTTP request. Therefore, to fixate a victim's session ID, the attackers need to first store the SID cookie in the victim's browser (in the domain of the targeted server). This is usually achieved through other vulnerabilities. For example, cross-site cooking exploits browser vulnerabilities to store information from one domain as a cookie in another domain. Cross-site scripting can also set cookies[2].

Using the session-fixation technique, attackers can also "donate" their current sessions to victims. In other words, the attackers intentionally let victims "hijack" their sessions. If the victims are unaware of the situation, and type sensitive information (e.g. credit card numbers) in the sessions, the attackers may be able to retrieve that information, as they have the control of the sessions; they can also monitor the victim's actions. The CSRF login attack is such an attack.

5.3 Session Confusion

Having been familiar with traditional client/server applications, users are used to the session concept in this type of application; they tend to perceive the session concept in web applications similarly. If the actual behaviors of sessions in web applications do not match this perception, users may be tricked into doing things that diminish security. We call this phenomena session confusion; it has led to a variety of attacks.

Signs of session termination. In a stateful framework, when a session terminates, it is quite obvious to the users, because users can see that the TCP connection is closed, the process has ended, etc. Session termination is quite obscure in web applications. When users switch to another web site (i.e. starting a new session with a new web site), it appears that their sessions with the previous web site has ended. This is not true; the session is still active, unless the user kills the browser or the server expires the session. Therefore, if the user switches into a malicious web site, the site can launch cross-site request forgery attacks on the user's account with the previous web site.

Session switching. In web applications, page redirecting is quite common. Users can be redirected from one page to another page or even to another site. If the redirection is to another site, users are basically switched from one session (with one server) to another session (with another server). The switching is transparent by design, so users may not be aware of such a switch, especially if the target site looks similar to the previous web site. This becomes a source of phishing attacks.

[2] JavaScript can be used to set victim's cookies, but it is not necessary; HTTP `<meta>` tag's `Set-Cookie` option can also instruct browsers to set cookies.

Some web applications fail to validate the target URLs when redirecting users. In many cases, the URLs are provided by other users, who trick the vulnerable web applications into redirecting victims to a malicious web site, which looks similar to the original site. If the victims are unaware of such a switch, they may type in sensitive information in the malicious site. This type of attack is ranked No. 10 in the OWASP Top-ten list [26]. Such an attack is difficult or impossible in the stateful client/server framework, because switching from one session to another is quite noticeable.

5.4 Extra Efforts to Preserve Session Integrity

To preserve the integrity of sessions, web developers must implement security measures in their applications. Several techniques are commonly used. A basic technique is to use long pseudo-random numbers as session identifiers [25], making it difficult for attackers to predict the number. This is already supported by several web application frameworks, such as PHP and Java Servlets. An additional layer of defense is to bind each session to an IP address [20]. If an HTTP request's originating IP address differs from the IP address tied to the session, the request will be rejected. This approach basically emulates the binding feature of the session in the stateful client/server framework.

Another common guideline is to regenerate session identifiers after user authentication [20]. Several web applications create a session even before users are authenticated, depriving attackers the chance to launch the session fixation attack. Using this method the application can make sure that an attacker does not have access to the session after user are authenticated.

In addition, web applications may also take additional measures to ensure that each request indeed originated from a legitimate web page. For example, web applications may add a secret token to each HTML form that they create, and check the presence of the token on each incoming request, ensuring that the request is initiated from within the page, not outside (e.g. cross-site request forgery). Moreover, some web applications use the HTTP Referrer or Origin headers to validate the source of requests.

Another common practice is to use short-lived sessions for important transactions [25]. For example, an online-retailing application can use a short-lived session for each checkout transaction. These sessions expire immediately at the end of each transaction. It is difficult for attackers to take advantage of these sessions because an attacker has to trick users or browsers into making a malicious request while the session is alive.

6. OUR POSITIONS

6.1 Mapping to OWASP Top-ten List

To summarize our discussion, we map the vulnerabilities on the OWASP Top-ten list [26] to the causes discussed previously (Figure 5). Six of the top-10 vulnerabilities can be mapped, making them traceable to the stateless nature of the Web. For the other four, we were not able to trace them to the stateless nature. After taking a closer look at these vulnerabilities, we found that they are not unique to web applications; they are also quite common in traditional applications. The mapping clearly supports our claim that the stateless nature of the web infrastructure is one of the root causes of the security problems on the Web.

OWASP Top-10 List	**Cause**
A1: Injection	-
A2: Cross-Site Scripting	Access Control
A3: Broken Authentication & Session Management	Session Integrity
A4: Insecure Direct Object References	Data Integrity
A5: Cross-Site Request Forgery	Access Control
A6: Security Misconfiguration	-
A7: Insecure Cryptographic Storage	-
A8: Failure to Restrict URL Access	Control Integrity
A9: Insufficient Transport Layer Protection	-
A10: Unvalidate Redirects & Forwards	Session Integrity

Figure 5: Mapping Vulnerabilities to Causes

6.2 Weakened TCB

We would like to go a step further to answer why the stateless nature can cause so many problems for stateful applications. As we can see from our discussions, a stateful framework has the ability to preserve trust states and code/data/session integrity. The preservation is achieved by the underlying operating systems, i.e., by the underlying Trusted Computing Base (TCB). When we chose a stateless framework for the Web, the ability to preserve those features was lost. Therefore, by choosing statelessness, we have chosen a TCB that is weaker than a stateful TCB. Probably, the pioneers who designed the web infrastructure have never thought about this consequence.

The Web enjoys a performance boost thanks to its stateless nature, i.e., the system underneath web applications avoids the responsibility of maintaining states, and thus improves performance. However, in the words of Robert Heinlein, "There Ain't No Such Thing as a Free Lunch;" now, we know how we pay for this: security. In some sense, we failed to anticipate the unexpected consequence of being stateless: a weakened TCB.

Faced by a weakened TCB, web application developers have to pick up the slack. They have to implement extra security mechanisms in their program logic to make up for the missing pieces in the underlying TCB. Such implementation is reflected in two aspects: (1) extra access control mechanisms to offset the inadequacy of the underlying same-origin and same-session policies; (2) extra mechanisms to protect the integrity of their control flow, data flow, and sessions. As we have discussed, mechanisms for these purposes have been developed and deployed. Unfortunately, requiring application developers to make up for the weakened TCB is problematic; there are three main problems.

First, there is an awareness issue: developers need to be aware that their TCB is now weakened, so they should not treat web applications the same as they do to stateful applications. Unfortunately, many developers are not aware of the weakened TCB, and many vulnerabilities in the Web are caused by such ignorance. We should not just blame developers for their ignorance; we should blame those who developed the web infrastructure, blaming them for putting such a new and unfamiliar burden on developers. Had the web infrastructure had a similar TCB like the stateful framework, this would not have caused so many security problems.

Second, there is an implementation issue: web application developers essentially need to implement some part of the TCB in their programs. As we know, implementing a TCB is not something that an average programmer can do, because the program logic for the TCB is quite complicated

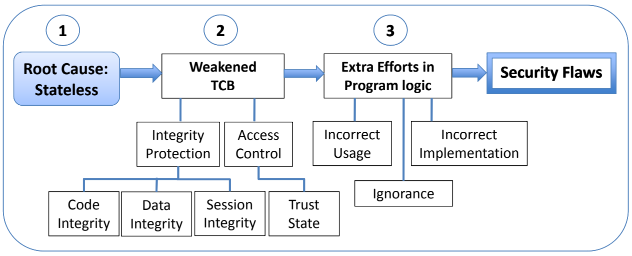


Figure 6: Root Cause and Chain Effects

and error-prone, requiring a strong background in security. Forcing unqualified developers to do such a job is an invitation for vulnerabilities. Traditional software development, although having its own problems, has much better TCB support, which provides systematic mechanisms to preserve trust state and to enforce code/data/session integrity. This may explain why the percentage of vulnerable web applications is much higher than general software.

Third, there is a usage issue: one may argue that since many applications have implemented those security mechanisms, other applications can simply reuse them. Indeed, companies like Facebook, Google, and many banks have put a lot of resources into securing their web applications. Unfortunately, because the security mechanisms are implemented within the program logic of their web applications, they are tightly coupled with the applications. Even if the code is open source, it is not easy to just use them directly. Other applications might reuse their ideas, but not their code. Moreover, unlike the TCB, which is meant for others to use, the security mechanisms implemented in those web application are not designed for others to use; therefore, they are not general enough. These situations make it difficult to use existing implementations correctly.

In summary, we believe that the stateless nature of the web infrastructure has led to a weakened TCB for the Web, forcing developers to implement extra protection mechanisms in their applications. Mistakes made during such an effort, or the lack of such an effort, have led to many of the security problems on the Web. We depict our positions in Figure 6.

6.3 Potential Solutions

Understanding the root causes and the chain of effects can lead us to better solutions. From the chain-effect diagram in Figure 6, it is quite clear that potential solutions mostly fall into three places: locations marked with 1, 2, 3 in the figure. Let us examine each of these locations.

First, we can fix the problem at the root (i.e. Location 1). This can be done by either turning the Web into a stateful system, or making web applications stateless. Neither approach is acceptable and practical. The former approach will cause significant downgrading of server performance, making it resistant to acceptance by industry. The latter approach is incompatible with the needs of most web applications. For example, an online-retailing application has to be stateful to track the contents of its customers' shopping carts.

Second, we can fix the problems that directly cause the security flaws. (i.e. Location 3). To achieve that, we can develop methods, reusable libraries, and tools to help developers make up for the deficient TCB, reducing their errors. We can also improve our education and training, turning all web application developers into security experts. These ideas sound familiar, as they are exactly what we have been doing for the last 15 years, and we are still facing the fact that over 80% of web applications are vulnerable. If we have not been so successful in this approach, it will be hard to believe that we can do better in the future, given the fact that Web applications are getting more and more sophisticated with the emergence of Web 2.0, HTML 5, and other advanced web technologies.

Third, we can fix the insufficient TCB at Location 2, so the web infrastructure can provide systematic security supports similar to those in the stateful framework, i.e., preserving trust states and the integrity of code, data, and session. With this support at the TCB level, developers will be liberated from those complicated and error-prone tasks; this can potentially lead to a significant reduction in the number of security flaws.

We strongly believe that the third approach is what we should take, although most existing work has taken the second approach. If the fundamental problems with the TCB are not fixed, even if we can fix a problem specific to certain attacks, new attacks will keep coming up, because the Web is still evolving, with new features being added regularly. We also acknowledge that fixing the TCB correctly is not an easy job; it requires a well thought-out design. While we are working on this direction with some preliminary results [7, 11, 22, 30], there is still a long way to go. We hope that this position paper can convince more researchers to join the pursuit.

7. SUMMARY

When deciding to make the Web stateless, developers of the web infrastructure failed to analyze the security consequence of such a decision; they failed to realize that being stateless would lead to a weakened TCB. No effort was made by the infrastructure developers to enhance the TCB up to a level that is comparable with the TCB of stateful frameworks. As a result, the burden of bridging the gap was left to web application developers, who have to bear the responsibilities of providing security support within their code. Average developers are not qualified to bear such responsibilities, and often make mistakes that lead to security vulnerabilities. That, we believe, is the main cause of the many security flaws in web applications.

8. REFERENCES

[1] Caja. `http://code.google.com/p/google-caja/`.

[2] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88, New York, NY, USA, 2008. ACM.

[3] P. Bisht and V. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks . In *DIMVA 2008*.

[4] D. Crockford. ADSafe. `http://www.adsafe.org`.

[5] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities inweb applications. In *Proceedings of the Eighteenth Usenix Security Symposium (Usenix Security)*, Montreal, Canada, 2009.

[6] K. Donald, E. Vervaet, and R. Stoyanchev. Spring Web Flow - Reference documentation. `http://static.springsource.org/spring-webflow/docs/1.0.x/reference/index.html`, 2007.

[7] W. Du, X. Tan, T. Luo, K. Jayaraman, and Z. Zhu. Re-designing the web's access control system (invited talk). In *Proceedings of the 25th Annual WG 11.3 Conference on Data and Applications Security and Privacy*, Richmond, Virgina USA, July 11-13 2011.

[8] M. Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure javascript subsets. In *Proc. of Network and Distributed System Security Symposium, 2010.*

[9] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2009.

[10] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW 2006.*

[11] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS)*, Genoa, Italy, June 21-25 2010.

[12] K. Jayaraman, G. Lewandowski, P. G. Talaga, and S. J. Chapin. Enforcing request integrity in web applications. In *Proceedings of the 24th annual IFIP WG 11.3 working conference on Data and applications security and privacy*, DBSec'10, pages 225–240, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW 2007.*

[14] M. Johns and J. Winter. RequestRodeo: Client-side protection against session riding. In F. Piessens, editor, *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pages 5 – 17, May 2006.

[15] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *In Proceedings of the Second IEEE Conference on Security and Privacy in Communications Networks (SecureComm)*, pages 1–10, 2006.

[16] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *CCS 2007.*

[17] F. Kerschbaum. Simple cross-site attack prevention. In *SecureComm 2007.*

[18] D. M. KIENZLE and M. C. ELDER. Final Technical Report: Security Patterns for Web Application Development. `http://www.scrypt.net/~celer/securitypatterns/finalreport.pdf`, 2002.

[19] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *ACM SAC 2006.*

[20] M. Kolsek. Session Fixation Vulnerabilities in Web-based Applications. `http://www.acrossecurity.com/papers/session_fixation.pdf`.

[21] B. Livshits and U. Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *PLAS 2007.*

[22] T. Luo and W. Du. Contego: Capability-based access control for web browsers. In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, Pittsburgh, PA, June 22-25 2011.

[23] L. A. Meyerovich and V. B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy*, pages 481–496, 2010.

[24] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2009.

[25] G. Ollmann. Web Based Session Management: Best Practices in Managing HTTP Based Client Sessions. `http://www.technicalinfo.net/papers/WebBasedSessionManagement.html`.

[26] OWASP. The ten most critical web application security risks. `http://www.owasp.org/index.php/File:OWASP_T10_-_2010_rc1.pdf`, 2010.

[27] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical prevention of large-scale data leaks. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[28] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID 2005.*

[29] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 1975.

[30] X. Tan, W. Du, T. Luo, and K. D. Soundararaj. Scuta: A server-side access control system for web applications. Technical Report SYR-EECS-2011-09, Syracuse University - Department of Electrical Engineering & Computer Science, July 2011.

[31] A. Vance. Times web ads show security breach. `http://www.nytimes.com/2009/09/15/technology/internet/15adco.html`.

[32] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 173–186, New York, NY, USA, 2009. ACM.

[33] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS 2007.*

[34] WhiteHat Security. Whitehat website security statistic report, 10th edition, 2010.