# Concurrency Attacks in Web Applications

Scott T. Stender

Alexander G. Vidergar



scott@isecpartners.com

av@isecpartners.com

## Introduction

Web application frameworks are built according to a number of design requirements. Frameworks should scale well, encourage developer productivity, be consistent programming best practices, and provide a secure platform for the application to stand on, to name a few.

Of course, each web application framework emphasizes these and other requirements according to their own philosophy. No framework can meet all of these requirements, so each framework is an expression of a sensible balance between them.

In our experience, the most commonly-used web application frameworks choose performance and scalability as the first, most important design criteria. Second, following closely after, is developer productivity. A design that seeks to maximize these criteria must trade with other criteria, and security is often the loser.

It is our opinion that today's common high-performance, object-oriented web application frameworks encourage programming practices that make managing concurrent access to shared data difficult to accomplish securely. Other classes of security flaws manifest themselves regularly when programming practices encourage those flaws; concurrency-based flaws are no different. It is our hope that with more awareness and better testing techniques that these flaws can be identified and eliminated.

### *Concurrency*

Concurrency refers to the set of polices and mechanisms that enable one or more threads of execution to operate simultaneously[1]. A number of attributes must be present in order to make this possible:

- *Infrastructure* – True simultaneity occurs when multiple processors are available, whether they are on a single machine or distributed across a network. The key is independent processors working cooperatively
- *Coordination* – Operating systems provide coordination between concurrent threads of execution by providing synchronization primitives and handling resource management, including scheduling of execution. Distributed systems require a similar architecture, often provided by operating systems, server applications, or dedicated transaction management services.

- *Programming Support* – Most commonly-used programming languages provide similar models for managing threads of execution and synchronization. These abstract the underlying complexity of services provided by underlying infrastructure and coordination facilities. Though simplified, programming for concurrency remains a difficult art.
- *The Program* – All of the other aspects can exist in a vacuum; they can be present and a programmer can still choose to execute their code in one long, sequential series of function calls. In order to take advantage of the promise of concurrency, developers must design and implement their software with an understanding of the underlying complexity.

The developer of a concurrent application has a difficult job: They must understand the underlying complexity of a concurrent system, work with the limited APIs and technologies at their disposal to build safely-concurrent systems, and do so with enough time left to make a functional system.

### *Concurrency Hazards*

Most literature for managing concurrency in the context of a particular language covers two areas: making the application perform well and avoiding hazards. Of course, a poorly-designed concurrency strategy will not deliver better performance or scalability. Though this could have Denial of Service impact, this paper will focus more on the security implications of concurrency hazards.

The concurrency hazards of most interest for security practitioners are those that have to do with inadequate synchronization of data access. As with most classes of security flaw, the goal of the attacker is to use flaws in the system to manipulate sensitive data. Inadequate or incorrect synchronization of data access and manipulation offers just that possibility.

## Concurrency Flaws

Concurrency Flaws exist when a concurrency hazard, such as a race condition or deadlock, exists in an application and carries a security impact. Such bugs may go unnoticed through functional testing, may only appear during boundary conditions, and/or may only be triggered during particularly heavy load. These types of flaws will result in unpredictable behavior that is difficult or time intensive to identify and fix.

### *Identifying Candidate Transactions*

Any data that is concurrently modified by multiple threads can provide a hazard for concurrency in a multithreaded environment. In many cases resources can be shared harmlessly between threads. However, resources which have conditional creation, modification, update requirements and are available for concurrent access should be considered a source of a potential flaw. Identifying which assets are accessible concurrently and then further scrutinizing which are used for sensitive operations can identify potential sources of concurrency flaws.

Once assets are identified, one must identify actions that manipulate those assets. Any action which changes the state of candidate assets is a potential candidate for a concurrency flaw. In particular, if the new state of the object prohibits the same action from occurring again or alters the conditions in which that action is allowed to occur.

- *Transferring Funds* - In a trivial bank example where an account object contains a balance and transfers can take place only if the transfer is less than or equal to the balance, the balance information becomes the control. If the entirety of the balance is transferred, then no additional transfers can take place. However, if a previous state can be accessed (one with a nonzero balance), the transfer action is completely acceptable. Thus a race condition exists in the transfer action and the update of the amount variable in the object.
- *Access Control Changes* – Authorization policies define the abilities of authenticated users within a system. Alterations to authorization polices can be susceptible to race conditions before changes are fully perceived by the system.
- *Digital Shrinkwrap*- Situations where redemption of prizes, one-time use objects, time-limited use objects, on-first-use objects, etc. where the intention is to allow controlled use of the object may be vulnerable or susceptible to concurrency flaws.

The increasing complexity of applications and the near-endless possibilities of managing sensitive assets introduce opportunities for concurrency flaws to manifest themselves. Although not all flaws will depend on controls, nor on the categories identified here, these offer an introduction to the potential hazards in concurrency transactions

## *Concurrency Flaws in Web Applications*

Concurrency is particularly difficult for web application programmers. The design model of a web application fundamentally is to allow as many concurrent users as possible and to respond to them as quickly as possible with information requested by those users. The demands of web applications in fact are so great that a single system cannot sufficiently service all users and thus load balanced, distributed, multi-database systems are implemented to further satiate the demands for more information faster.

Several forces act on web applications to encourage the presence of concurrency flaws:

### Developer Productivity

The complexity of developing and deploying a multi-threaded multi-tiered web application has been reduced in modern web application frameworks, web servers and database systems. This is both a great asset and a great risk. When the bar of entry for deploying a complex system is lowered, there is additional risk that naïve implementations of web applications will lead to flaws. Given the subtlety of concurrency flaws, it is likely that these would escape notice in such systems.

## Performance

Protection against concurrency flaws requires the locking of assets that could be manipulated by multiple sources. By default, most web application frameworks, databases, and operating systems attempt to process as many requests as quickly as possible. The action of locking a resource implies that other threads of execution must wait for that resource before continuing, which in turn affects performance. Thus, application design must consider shared resources: the performance of an application is in opposition to the safe access of the resources it manages.

## Encapsulation

One of the tenets of Object-Oriented Programming is data encapsulation. It encourages the engineering practice of abstracting low-level data and actions into higher-level objects. This allows predictable actions to be performed, checked, and extended on data objects, which is a big win for code re-use and developer productivity.

However, data encapsulation implies that the data is hidden from developers wishing to interact with the software at the object-level. The purpose of encapsulation is, in fact, to free developers from such drudgery. However, sensitive shared data must be protected, and a developer relying on an opaque object may not realize that sensitive data needs to be protected.

Of course, Object-Oriented Programming can help manage concurrency. Indeed, patterns for using the strengths of OOP to manage concurrent access within an object's members are well-understood and widely-used. However, encapsulation ends at the scope of the object, and functions that tie object methods into a single transaction need to manage synchronization. For example, an Account object may provide perfect thread safety for each of its member variables when accessed through its defined methods. However, a "TransferFunds" function that ties together multiple balance queries and updates on that object will be subject to concurrency flaws if it does not manage synchronization itself.

In this particular case, one would hope that the need to synchronize access to the asset is obvious to the developer, though not all such cases are as clear. When naïve developers take dependencies on opaque objects, the principle of encapsulation can be contrary to security.

## Inconsistency

As consultants, we work with a large number of technologies over the course of a year. We find that web application frameworks are terribly inconsistent in what they consider thread-safe. From a developer's perspective, this means that moving from between frameworks requires a rather error-prone context switch. They are likely to find that a reasonable assumption of thread safety is no longer the case when frameworks are modified or updated to the latest version. The following examples illustrate the degree of variation:

- *ASP.Net* – Requests for a given session are serialized, so session variables are thread-safe by default.
- *Java Servlets* – HttpSession, including associated variables are not thread-safe
- *Struts 1.x* – Actions are singletons and thus prone to thread-safety issues. The documentation emphasizes the need for the application designer to develop action classes to be thread safe.
- *Struts 2.x* – New instances of Actions are spawned for each request and are thread safe on a per-instance basis.

Those are just the common variables used by web app frameworks. Static classes, widely-scoped variables, and access to back end systems are all shared resources that need to be synchronized as well.

# Identifying Concurrency Flaws

The identification of concurrency flaws is a challenging task, perhaps even more challenging than developing concurrent software. To find a concurrency flaw, one must either understand that a piece of data happens to be shared by concurrent consumers and recognize the lack of synchronization between them or generate tests that stress a concurrent system and identify signs of flaws.

It is assumed that, based on the previous section, one is able to identify sensitive transactions to be analyzed for flaws. A general procedure for finding flaws in such transactions follows:

## *White Box Analysis*

White Box analysis assumes that the reviewer has full access to system information, including specifications, code, and perhaps access to system engineers. The techniques available during White Box analysis is a superset of the techniques available to Black Box reviewers, giving White Box reviewers the ability to pick and choose the most appropriate approach to finding flaws.

Where possible, we suggest starting with the procedures discussed in the White Box analysis and augmenting them with those described in the Black Box analysis section.

### Identify Assets

The first step is to identify the data associated with the sensitive transaction to be tested. These are the assets that the attacker wants to manipulate. Such data could be stored in a number of places, in an HTTP session variable, in a database table, or accessed via a SOAP request, for example.

### Identify Consumers and Synchronization Mechanism

Once assets have been identified, it is important to find all known consumers of the data and what, if any, mechanisms are used to synchronize access to it. What constitutes "appropriate synchronization" depends primarily on the means by which the data is stored and accessed.

For data stored in process memory, "consumers" might be defined as the set of threads that call a well-known set of functions to interact with said data. In this case, one can use synchronization primitives provided by the operating system and/or programming environment to manage data access. Code review should focus on ensuring that data access is accomplished through a common interface and that it provides appropriate levels of synchronization.

For data stored in a database, "consumers" might be the set of front-end web servers in a farm together with a back-end management server. In this case, one has to evaluate the distributed transaction processing mechanism used to synchronize access. In the case of database access, one can rely on the inherent support for transactions available in most modern databases. However, distributed transaction processing is a problem that stretches beyond database access. For other types of systems, it is likely that similar, technology-specific, synchronization mechanisms will be used. See the "Best Practices" section of this document for examples of such mechanisms.

For both general classes, concurrent threads and concurrent machines, a combination of code review, documentation review, and interviews with system engineers would be the best means of identifying synchronization mechanisms.

### Verify Proper Synchronization

Once all consumers of shared data have been identified along with their use of synchronization mechanisms, it is time to verify that the data in question is properly protected. First, the nature of the data manipulation should be grouped into sensible transactions.

For example, transferring funds from one bank account to another can be decomposed into several accesses of the key pieces of data – the balance on the source and destination account. One must first read information about each account, perform validation of the transfer request, and then write information about each account – a total of four data accesses. In order to correctly perform this transaction, each access must be performed in the scope of a single transaction. Otherwise, modifications to the source or destination account balances performed mid-transfer could be overwritten.

In order to pass review, it is necessary to verify not only that data access is properly synchronized through common access routines. It must be synchronized properly through all known access routines. Ideally, a common synchronization mechanism exposed through a common interface would be used for simplicity's sake. Where multiple synchronization methods are used, additional review should be performed to verify that each method provides equivalent protection to shared data.

## *Black Box Analysis*

Black Box testing, generally speaking, means that a limited subset of information is available to the reviewer. For the purposes of this paper, we will assume that the Black Box reviewer has access only to a system to test, not to code, binaries, or documentation of the system. Black Box analysis necessarily takes a test-centric approach for this reason.

Identifying concurrency flaws through testing is difficult. A single test cannot conclusively identify the presence of a concurrency flaw. Instead, systems must be rigorously tested under stress to ensure transactional integrity. The following serves as a general testing method for finding such flaws:

## Build Test Cases

Testing for concurrency flaws is similar to any other test: you have to define the test steps, determine how tests can be measured for success or failure, and execute them.

To build your tests, first use the guidance in the "Identifying Sensitive Transactions" section in this document to create a set of transactions to be tested. For example, we might determine that the "Transfer Funds" feature of the bank described earlier is a good target for attack.

Next, use a web proxy tool to capture the unique request or requests required to exercise each transaction and their expected outcome. In the same example, we might capture HTTP POST requests to the "TransferFunds.aspx" page so that they can be replayed during the test. The expected behavior of such a transaction would be that the sum of the source and destination accounts always remains constant. When this no longer holds, it is an indication that something is amiss in transaction processing.

## Perform Concurrency Tests

The next step is to perform tests in an attempt to demonstrate the concurrency flaw. Often, simply demonstrating that directed testing can put the system into an inconsistent state is sufficient to demonstrate the presence of a flaw.

There are strategies to consider when building a concurrency testing suite:

First, one must consider where the asset might be managed. If, for example, the asset is managed in a session variable, all tests should be performed using the same session identifier. Similarly, if assets are available only across authenticated sessions, each request should be made using a different session identifier. Finally, if the shared data is stored on a back-end server and accessed via one of several front-end servers, tests should target each of the front-end servers for maximum impact.

Second, one must architect their tests to maximize the possibility of triggering the flaw. Concurrency attacks, by and large, occur with a certain probability. One can often increase the probability by stressing the server. This makes a certain amount of sense: by increasing the number of requests the server must serve, the server will likely increase the number of threads it manages. The presence of additional threads, especially those performing the same transaction, increases the probability that one of those transactions will be interrupted mid-execution.

In our experience, concurrency flaws can commonly be identified using a single, high-performance client library for repeatedly executing requests that trigger transactions. We have created a freely-available open-source tool, SyncTest, to help perform this kind of testing.

### Validate Proper State

Once the concurrency testing suite has completed, the final step is to verify that the presumed state is consistent with the expected result. Per the "Transfer Funds" example above, one would verify that the sum of the balances for the source and destination accounts would be equal before and after the tests.

## Best Practices

The concept of concurrency has been a research subject and practical consideration of developers for decades. There has been a sizeable collection of books, papers, and how-to documents published on the subject, including a number of books that focus on well-tested patterns for addressing concurrency safely. There is no single API or approach that fits every concurrency challenge, though there is a general tool that is effective in almost every case: proper synchronization. To attempt to enumerate an exhaustive list of proper techniques to apply to each challenge is impossible. However, the following resources should be able to help any developer who seeks to address the class of flaws identified in this paper.

### *System-Level Synchronization*

If the data that is managed is limited to a single application or to multiple applications running on a single system, one can use language and OS-level synchronization primitives to manage concurrent access to resources. Most programming references for languages discuss APIs for managing threads and synchronization objects. However, the cursory coverage given in such references is rarely sufficient to address more difficult problems in concurrency and synchronization. For in-depth language or operating system references, we would recommend picking up a book focused on the subject to get a full understanding of the capabilities and limitations of the technology. For a broader perspective that covers how to architect systems that both perform well and safely manage concurrency, we recommend *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, referenced in the endnotes.

### *Distributed Synchronization*

Synchronization across machines is a much more difficult matter. As mentioned elsewhere in the paper, most common databases incorporate transaction support and can be used to manage synchronization of data access.

However, there are myriad systems to synchronize, and one cannot simply assume that databases will always be present to help us. Several standards exist for addressing this problem across networked systems, the most commonly-used of which is the X/Open XA[2] standard. Such standards are often adopted by dedicated transaction management servers, operating system-provided transaction managers, and network protocols designed to support transactional integrity. Again, an exhaustive discussion would be well beyond the scope of this paper. We recommend *Principles of Concurrent and Distributed Programming, 2ed*[3] for a theoretical overview of the requirements for distributed synchronization and a review of your system's technical documentation for application-specific support and details.

## Acknowledgements

The authors would like to thank Jesse Burns and Brad Hill of iSEC Partners for their commentary and guidance during the research that supported this whitepaper. Any errors in this whitepaper are our own, but without their help there would undoubtedly be more.

---

[1] Schmidt, Douglas, Michael Stal, Hans Rohnert, and Fank Buschmann. Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects. Chichester: John Wiley & Sons. 2000

[2] The Open Group. "Distributed TP: The XA Specification" 6 Jun. 2008. <http://www.opengroup.org/bookstore/catalog/c193.htm>

[3] Ben-Ari, M. Principles of Concurrent and Distributed Programming. 2nd ed. Essex: Addison-Wesley, 2006