Concurrency Attacks in Web Applications

Presented at Black Hat USA 2008

Scott Stender, Principal Partner, iSEC Partners.

Alex Vidergar, Security Consultant, iSEC Partners

iSEC Partners

https://www.isecpartners.com

Introduction

- Who are you?
  - Security Consultants at iSEC Partners
  - Work in our application security consulting practice
  - Based in Seattle and San Francisco

- What is this talk about?
  - Identifying and exploiting common conditions that arise from concurrency in web applications

- Why should I care?
  - Not every application needs to for security purposes
  - At some point, you will probably be involved with one that does!
  - Attacks against concurrency are well-known in many application types, but poorly-covered in the web app world. No more!

Introduction to Concurrency

- Concurrency refers to the ability for a system to run multiple threads of execution “simultaneously”

•Single Application

- •Time division across threads of execution
- •Increasing the responsiveness of expensive tasks
- •Scalability

•Across Systems

- •Web-sever can process thousands of requests at once
- •Database multiple sessions managed on same source

Introduction to Concurrency

- What does it take to handle multiple threads of execution?
  - A way to define them: A “thread main” function
  - A way to manage them: Operating System APIs
  - A way to share data between them: Shared memory
  - A way to synchronize access between them: Operating System objects
- Sound complicated? It is!
  - Designing, implementing, and testing scalable applications is a subject of many-a-research paper, book, presentation, and consulting!

Introduction to Concurrency

- A Simple Web Server:
  - Accepts connections on ports 80 and 443
  - Has to read request data, process it, build response, send it
  - Imagine if we did this with a single thread of execution!

- Designing a Simple Web Server
  - One thread of execution handles accepting connections
  - Worker threads accept a handle to that connection and handle processing
  - Web Application Frameworks give a pretty API to developers for custom processing

Introduction to Concurrency

- Web Application Frameworks
  - Define an interface for consumers of the framework
  - The function called by the framework includes request context: cookies, global application variables, session variables…
  - Most popular frameworks today are object-oriented:
    - Session[“UserName”] = Request.Parameter(“UserName”)
    - Response.Write(“<html>…</html>”)
    - Response.Send()

- It is easy and intuitive to have an interactive web page up and running within minutes!

Security Implications

What happens when those threads execute simultaneously?

- The answer is “it depends”
- The general answer is “Access to shared resources needs to be protected”
- That’s easy – I bet the framework takes care of that!

Security Implications

Struts, *Programming Struts, O'Reilly*

- …keep in mind that the HttpSession is not synchronized. If you have multiple threads reading and writing to objects in the user's session, you may experience severe problems that are very difficult to track down. It's up to the programmer to protect shared resources stored in the user's session.

Hibernate, *the Hibernate Reference-Transactions & Concurrency*

- "Uses directly JDBC… without adding additional locking behavior. We highly recommend you spend some time with the JDBC, ANSI, and transaction isolation specification of your database management system."

Struts 2, *Apache Struts2 Documentation, Migration Guide*

- "Struts 2 Action objects are instantiated for each request, so there are no thread-safety issues."

Java, *the Java Language Specification*

- Thread safety synchronizes statements and methods however:
  "The Java language neither prevents nor requires detection of deadlock conditions."

Security Implications

- Consider an online bank's "Transfer Funds" feature

```
Function TransferFunds(src_acct, dest_acct, amount)
{
   src_amt = src_acct.balance
   if (src_amt >= amount)
   {
       src_acct.balance = (src_amt - amount)
       dest_acct.balance = (dest_acct.balance +
        amount)
   }
}
```

Security Implications

Thread 1 – xfer $1	Thread 2 – xfer $1	
`src_amt = src_acct.balance`		`src_amt = $10`
	`src_amt = src_acct.balance`	`src_amt = $10`
	`if (src_amt >= amount)`	
	`src_acct.balance = (src_amt - amount)`	`src_acct.bal = $9`
	`dest_acct.balance = (dest_acct.balance + amount)`	`dest_acct.bal = $11`
		`$src_amt = $10`
`if (src_amt >= amount)`		
`src_acct.balance = (src_amt - amount)`		`$src_acct.bal = $9`
`dest_acct.balance = (dest_acct.balance + amount)`		`$dest_acct.bal = $12`

Security Implications

- Once the dust settles – you are $1.00 richer!
- How might you protect against this?
  - Generally: the process needs to be considered a single transaction
  - Shared values, such as balance, should be locked during execution
- There are many options that we can consider:
  - Use thread synchronization primitives (mutex, semaphore, lock, etc.) to prevent Thread 2 from starting until Thread 1 has finished
  - Specify the whole thing as a transaction to be handled by the database

Security Implications

- Concurrency flaws, as a class of attack, are commonly known
  - Race conditions, TOC-TOU, state attacks, etc.
  - Low-level application and kernel developers live by this stuff
- However…there is little guidance in the area of web applications
- And…common web application frameworks designed for ease of use punt this to web developers with little “aside” comments
- Finally…testing for these in web apps is particularly tough
- Our opinion…this is an under-tested class of flaw in web apps

Security Implications

- What makes a bug in synchronization a security flaw?
  - High priority flaws have a transaction that involves a stateful asset, like a bank account
  - Control parameters that change, such as authentication credentials, single use redemption tokens, etc.
  - Reliability also can take a hit when state gets messed up, leading to Denial of Service or corrupted data
- The key takeaway – if your application has stateful assets, watch out!

Design Considerations

- Frameworks themselves protect against concurrency flaws
  - .NET – Multithread safety on local and session variables
  - Struts2 relies on the Java protections
    - Concurrent Modification Exceptions, Synchronized methods
    - Each action has own instantiation of objects
- Framework + Framework
  - How reliable is a distributed load balanced application across servers?
- Framework + Database
  - What about when an external data source is added?
- Framework + Framework + Database
  - Distributed applications to a single database?
- Framework + Framework + Database + Database …
  - Distributed applications to distributed data stores?

Design Considerations

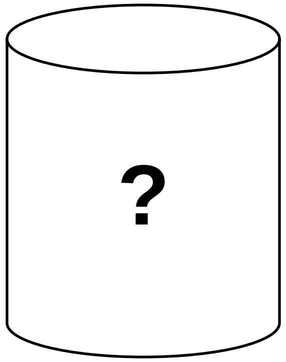
How to fix this at the application: an evolution

- x Initial Design
  - Query database to service user request for list
  - Allow user to modify information and then POST an update
- x Check for dirty information
  - Before submitting check to see if db is at known state
- x Put the requests *really* close together
  - Maybe even on the same line!
  - I can’t repro it now!
- ? Use a global lock
  - Acquire the lock before the transaction, release it at the end
- ✓ Encapsulate end-to-end transactions in a scoped lock
  - Remember, only you can prevent deadlocks
- ✓ Handle it at the database…

Design Considerations

How to fix this at the database: an evolution

- x Initial Design
  - Query database to service user request for list
  - Allow user to modify information and then POST an update
- x Check for dirty information
  - Before submitting check to see if db is at known state
- x Update only with known information
  - submit an update selecting row from database of known state
  - UPDATE balance WHERE accountID=457 AND balance=1.00;
- ? Limit all threads to single DB connection
  - Share one pipe to the database for all instances of action class
- ✓ Use database transactions support
  - Database configured to support transactions, result sets, etc.
  - Remember your isolation levels!



Design Considerations

Database Considerations

- For DB transactions a session consists of atomic parts
  - [lock], read, update, promptly commit, [unlock]
  - The prompt part of this is ruined due to user interaction
    - Read, [User Think Time], updated, then commit
      - Cannot always lock and application while waiting for user
- Well formed sessions however can be handled by the DB
  - Only one session can access these data elements at once
  - Warning: Some databases fetch multiple rows on queries for efficiency and may lock all of them, producing unexpected results
- If the DB cannot be locked for a whole session, then the session must be broken into multiple session for each part
  - By executing each action separately, the DB controls are broken and liability of data integrity is returned to the application

Design Considerations

Data Considerations

- Data **C**reation **R**eading **U**pdate **D**eletion requirements
  - What data must be available for CRUD operations?
- Data Access Requirements
  - Is it required to allow multiple users to access same data
- Time Sensitivity
  - Is there actions that may be affected by changes in data?
- Serviceability
  - Is failure to update due to concurrency locks an acceptable event
  - How will the user be affected by a failed attempt

Design Considerations

- The number of ways this can go wrong and be fixed is long
  - Think the set of applications ever written or could be written
- Just remember to keep the scope of the data in mind
  - Per-application
  - Per-system
  - Distributed system
- And use the right technology for the job!

Finding Concurrency Flaws

- If you have access to code and documentation, you might be able to go the formal approach
  - Identify all shared data
  - Identify all access to that data from across systems
  - Find data access that does not get synchronized
  - …5000 requests later…
  - Profit!

Finding Concurrency Flaws

- Not so fast…
- The technologies involved are numerous and complicated
- Before you file that bug, check out
  - OS synchronization primitives in code
  - Configuration of the database
  - Potential involvement of transaction coordination services

Testing For Concurrency Flaws

- Testing for concurrency/synchronization bugs is tough even for functional bugs
  - There is no single fire-and-forget test
  - If you have a single request being executed serially, you will never find anything (sound familiar?)
  - What you look for is discrepancies after performing load tests
  - You can add test hooks that will encourage context changes during execution

Testing For Concurrency Flaws

- Most stress testing is performed to identify a rough transactions-per-second metric
  - You may find speed and reliability, but you need targeting to find security flaws
- Consider what assets you need to protect:
  - Per-user session variables requires stress testing with the same authenticated user
  - Cross-user global variables and/or data sources requires stress testing cross users

Testing For Concurrency Flaws

- We have created a tool that can help with these kinds of tests
  - SyncTest, available at http://www.isecpartners.com/tools
- You have to identify the transaction to test
  - And associated requests
  - AND a means to check whether your attacks are successful
- We take care of making a ton of those requests for you!
- The default test simply creates a pile of requests to hit the server
- Open-source python means you can tweak the timing to your heart’s content

Acknowledgements

Many thanks are due to Jesse Burns and Brad Hill of iSEC Partners for their comments and support during the research that supported this project.

Thanks!

Concurrency Attacks in Web Applications

Q&A

scott@isecpartners.com

av@isecpartners.com