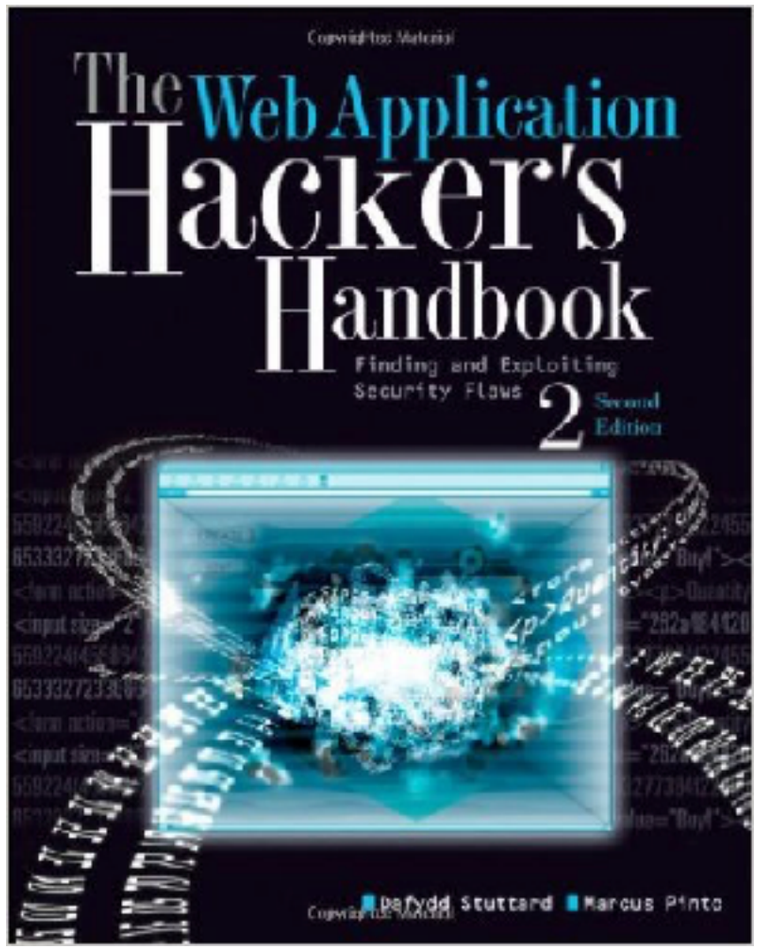
CNIT 129S: Securing Web Applications

Ch 11: Attacking Application Logic



Logic Flaws

- **No common signature, such as found in SQL injection**
- **Often an assumption a developer made**
- **Difficult to find and eliminate**

Real-World Logic Flaws

1. Asking the Oracle

- **"Remember me" function sets a permanent cookie**
- **Containing an encrypted string that contains**
- **Name, User ID, and volatile data to make it unique and unpredictable, including machine IP address**
- **Screen name also saved in encrypted form as ScreenName**

Assumption

- **It's OK to use same encryption algorithm to encrypt both cookies**

- **But user can control ScreenName**

- **And the app decrypts that cookie, showing the result on the screen**

The Attack

- **Copy the RememberMe cookie into the ScreenName cookie**
- **The app decrypts it and shows the result**

```
Welcome, marcus|734|192.168.4.282750184
```

- **Now change screen name to**

```
admin|1|192.168.4.282750184
```

The Attack

- **Log out, log back in, and copy the new ScreenName cookie to the RememberMe cookie**
- **Attacker is now admin!**
- **Encryption was 3DES and unbreakable, but it didn't matter**

Hack Steps

- **Look for items that are encrypted, not hashed**
- **With data from the user**
- **Substitute other encrypted values**
- **Try to cause an error that reveals the decrypted value**

2. Fooling a Password Change Function

- **Form for password change asks for**
  - **Username**
  - **Existing password**
  - **New password**
  - **Confirm new password**

2. Fooling a Password Change Function

- **Administrators have a form that can change any password, implemented by the same server-side script**

- **Administrator's form doesn't ask for existing password**

The Assumption

- **When a request comes in without an existing password, that indicates that it came from an administrator**

```
String existingPassword = request.getParameter("existingPassword");
if (null == existingPassword)
{
    trace("Old password not supplied, must be an administrator");
    return true;
}
else
{
    trace("Verifying user's old password");
    ...
```

The Attack

- **Submit a password change without any existing password**
- **Reset anyone's password**
- **This really happened in the AOL AIM Enterprise Gateway application**

Hack Steps

- **Try deleting each parameter, one by one**
- **Delete the name as well as the value**
- **Try it at each step of the process**

3. Proceeding to Checkout

1. Browse the product catalog, and add items to the shopping basket.
2. Return to the shopping basket, and finalize the order.
3. Enter payment information.
4. Enter delivery information.

- **Assumption**
  - **Users will perform steps in sequence**
  - **A user on the last step must have entered payment details**

The Attack

- **"Forced Browsing"**
  - **Circumvent controls that make the steps occur in sequence**
- **Proceed directly from step 2 to step 4**
- **Get product without paying for it**

Hack Steps

- **Try skipping stages, doing a single stage more than once, and doing earlier stages after later ones**
- **Stages may use different URLs or parameter values**
- **Guess assumptions and violate them**
- **Watch for interesting error messages**

4. Rolling Your Own Insurance

- **App lets users obtain quotes for insurance, and, if desired, submit an insurance application online**

- **It used a dozen stages**

- **1. Applicant submits basic information, and either preferred monthly premium or amount of desired insurance payout**

  - **App computes values the applicant did not specify**

4. Rolling Your Own Insurance

- **2. Across several stages, applicant supplies other personal details: health, occupation, pastimes, etc.**

- **3. Finally application is sent to an underwriter**

  - **Underwriter uses the same web app to review the details and decide whether to approve the application, or modify the initial quote to reflect additional risks**

4. Rolling Your Own Insurance

- **Each stage uses a shared component to process each parameter of user data**

- **Component parsed all data in each POST request into name/value pairs and updated state information**

Assumption

- **Each request will contain only the parameters requested in the current HTML form**
- **Developers did not consider a user who submitted extra parameters**

The Attack

- **Supply valid data at earlier stage**
  - **But then overwrite it with later requests resetting the same value**
  - **No validation was performed on the unexpected parameters**
- **Allowed an XSS injection that revealed personal information of other applicants**

The Attack

- **Purchase insurance at an arbitrary price**
  - **Replace monthly premium at later stages**
- **Force approval**
  - **Underwriter sets parameters in same web app to indicate disapproval**
  - **Attacker can set them, bypassing the actual underwriter**

Hack Steps

- **Take parameters from one stage, and add them to requests from another stage**

- **Take parameters used by one type of user and try submitting them as another type of user**

5. Breaking the Bank

- **App lets existing bank customers register for online banking**

- **Collects name, address and date of birth**
  - **But no PIN or any other secret**

- **Forwards request to back-end system**

- **Mails an application pack to the customer containing instructions, a phone number for activation, and a one-time password**

Assumption

- **Designers regarded this process as safe, with three layers of protection**
  - **Some personal data required to start the process to deter impostors**
  - **Secret one-time password sent by mail; difficult for attacker to steal**
  - **Customer required to call in and authenticate with personal information and selected digits from a PIN**

Data Structure

- **Customer information stored in database as this object**

```
class CCustomer
{
      String firstName;
      String lastName;
      CDoB dob;
      CAddress homeAddress;
      long custNumber;
      ...
```

The Attack

- **Same data object used for online banking and registration**

- **Account details shown on main e-banking page were generated from the customer number**

- **Main banking application required several levels of authentication and access control to access the data**

Attack Steps

- **1. Log in with valid credentials**
- **2. Using the authenticated session, go to registration function and submit a different customer's personal information**
  - **The app overwrites the CCustomer object with a new object relating to the targeted customer**
- **3. Return to the main application functionality and access the other customer's account**

Fundamental Flaw

- **Same database object can be written two ways**
- **1. Main banking function allows writing after strict authentication**
  - **These designers think the user is known**
- **2. Registration function allows writing without authentication**
  - **These users are unknown**

6. Beating a Business Limit

- **Financial personnel can transfer funds between company bank accounts and customers and suppliers**

- **Application prevents must users from performing transfers over $10,000**

- **Larger transfers require a senior manager's approval**

The Code

```
bool CAuthCheck::RequiresApproval(int amount)
{
    if (amount <= m_apprThreshold)
        return false;
    else return true;
}
```

- **Any transaction that's too large requires approval**

The Attack

- **Transfer a negative amount**
  - **Such as -$100,000.00**
- **No approval required because it's below $10,000.00**
- **Money flows in opposite direction**

Numeric Limits

- A retailing application may prevent a user from ordering more than the number of units available in stock.
- A banking application may prevent a user from making bill payments that exceed her current account balance.
- An insurance application may adjust its quotes based on age thresholds.

- **Try negative values at each step**

7. Cheating on Bulk Discounts

- **Users order software products**
- **Discount if a bundle of items purchased together**
  - **25% discount for buying antivirus, firewall, and antispam all together**

Assumption

- **Discount applied when items added to shopping basket**

- **Developers assumed that shopper would buy everything in the basket**

The Attack

- **Add every item possible to the basket**
  - **Get discounted price**
- **Remove unwanted items from basket**
  - **Discounted price persists**

8. Escaping from Escaping

- **Found in the web interface for a NIDS**
- **User-controlled input placed in an operating system command**
- **Developers understood the code injection risk**
- **Added backslash to escape these characters:**
- **; | & < > ' space newline**

The Attack

- **Developers forgot to escape the backslash itself**
- **Attacker enters**
  - **foo\;ls**
- **Application converts it to**
  - **foo\\;ls**
- **Which allows the ; to get through unescaped**

9. Invalidating Input Validation

- **Input validation system**
  - **SQL injection filter changes all quotes to double-quotes**
    - **Will be interpreted as literal quotes, not metacharacters**
  - **Length limit truncates all input to 128 characters**

Example

- **This input**

```
admin'--
```

- **Changes to this, which fails to bypass the login**

```
SELECT * FROM users WHERE username = 'admin''--' and
password = ''
```

The Attack

- **Submit a username of 127 a's followed by a single quotation mark, and password foo**
  - `aaaaa[...]aaaaa'`
- **App adds another ', but the length limit removes it**
- **This causes a SQL syntax error**
  - `SELECT * FROM users WHERE username = 'aaaaa[..]aaaaa'' and password = 'foo'`

The Attack

- **Submit the same username, and a password of**
  - **`or 1=1--`**
- **Query becomes this, bypassing the login**
  - **`SELECT * FROM users WHERE username = 'aaaaa[..]aaaaa'' and password = 'or 1=1--'`**
- **" is interpreted as a literal ', not a metacharacter**

Detecting This Error

- **Submit strings like this, look for SQL errors**

```
''''''''''''''''''''''''''''''''''''''" and so on
a'''''''''''''''''''''''''''''''''''''" and so on
```

- **Vulnerabilities occur when input passes through sequential validation steps**
- **One step can undo another step**

10. Abusing a Search Function

- **Application provides access to a huge archive of information**
  - **Accessible only to paying subscribers**
- **Provided powerful search engine**
- **Anonymous user can perform a query to see what's available**
  - **But must pay to read the found articles**

Assumption

- **User cannot get useful information from the search function before paying**
- **Document titles were typically cryptic, like**
  - **"Annual Results 2010"**
  - **"Press Release 08-03-2011"**
  - **Etc.**

The Attack

- **Query searches full text of documents**
- **Guess at contents, and deduce them from the number of found documents**
- **Like blind SQL injection**

```
wahh consulting
>> 276 matches
wahh consulting "Press Release 08-03-2011" merger
>> 0 matches
wahh consulting "Press Release 08-03-2011" share issue
>> 0 matches
wahh consulting "Press Release 08-03-2011" dividend
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover
>> 1 match
wahh consulting "Press Release 08-03-2011" takeover haxors inc
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover uberleet ltd
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover script kiddy corp
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover ngs
>> 1 match
wahh consulting "Press Release 08-03-2011" takeover ngs announced
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover ngs cancelled
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover ngs completed
>> 1 match
```

Real-World Application

- **Authors have used this technique to brute-force a password from a configuration file stored in a wiki**

- **With these searches**

```
Password=A
Password=B
Password=BA
...
```

11. Snarfing Debug Messages

- **App is new and buggy, so it puts out detailed error messages containing:**

- The user's identity
- The token for the current session
- The URL being accessed
- All the parameters supplied with the request that generated the error

Assumption

- **There's no important information in the error message**

- **Because the user can get all that data by inspecting requests and responses from the browser anyway**

The Flaw

- **Error message was not built from the browser's information**
- **It came from a stored container on the server-side**
- **Not session-based**
- **Error condition copies data to the container, and then displays information copied from that container**

Race Condition

- **If two users have errors at nearly the same time**
- **One user's data is copied to the container**
- **But then displayed to a different user**

Exploitation

- **This is even worse than the race condition**
- **Attacker polls error container URL repeatedly**
- **Log results each time they change, and get**

- A set of usernames that could be used in a password-guessing attack
- A set of session tokens that could be used to hijack sessions
- A set of user-supplied input, which may contain passwords and other sensitive items

12. Racing Against the Login

- **Robust, multistage login process**
- **Users required to supply several different credentials**
- **Authentication mechanism had been subjected to numerous design reviews and penetration tests**
- **Owners had high confidence in it**

The Bug

- **Occasionally a customer logged in and gained access to a different user's account**
- **This seemed random and non-repeatable**
- **Eventually the bank discovered that this happened when two users logged in at precisely the same time**
  - **But not reliably**

The Flaw

- **Application stored a key identifier about each newly authenticated user in a static, nonsession, variable**

- **This variable's value was read back an instant later**

- **If a different thread, processing another login, wrote to that variable in between, the account would change**

Race Condition

- **Application was using static storage to hold information that should have been stored on a per-thread or per-session basis**
- **This is called a "race condition"**
- **A brief moment of vulnerability**
- **To exploit it, attacker must "win the race"**

Avoiding Logic Flaws

- **Document every aspect of the application's design thoroughly**

- **So an outsider can understand every assumption the designer made**

Avoiding Logic Flaws

- **Require clear comments in source code documenting:**
  - **The purpose and intended use of each component**
  - **Assumptions made by each component about anything that is outside of its direct control**
  - **References to all client code that uses the component**

Avoiding Logic Flaws

- **During security review, reflect on every assumption made in the design**
- **Imagine circumstanced that violate those assumptions**
- **Focus on conditions that user can control**

Avoiding Logic Flaws

- **During security review, think laterally about:**
  - **Ways the app handles unexpected user behavior**
  - **Potential side effects of any dependencies and interoperation between code components and application functions**