Introduction to Logic Flaws

What is a Flaw in Code?

A software vulnerabilities are defects in the code that could be exploited. Security expert McGraw separates these into two categories: bugs, and flaws [1,2]. Bugs are implementation problems in software. For example: an off-by-one error, buffer overflows, and using unsafe methods. Bugs will only exist at the code level, can be discovered using scanners, and can typically be fixed on one line or in a localized area. It does not matter what the code is being used for.

Flaws, on the other hand, could be an issue with the design itself, or with how the design was implemented in the code. Flaws are typically related to application or business logic, or the architectural design. Examples of include recovery systems that fail insecurely, poor password requirements, poor access control, insufficient process validation, lack of encryption, and bad error handling (fail-open) [2-4]. It is difficult to scan software for logic flaws. To fix flaws, the logic or architecture may need to be redesigned, and it could affect multiple areas of code.

Several organizations maintain updated databases of different types of logic flaws. A taxonomy of business logic vulnerabilities is available in the Common Weakness Enumeration (CWE) by Mitre [5]. The Open Web Application Security Project (OWASP) has categories of flaws to address during testing [6].

Where Do These Flaws Come From?

The initial software or system requirements may not have included security requirements; therefore security was not addressed during development. This would lead to a higher number of bugs and flaws than applications that did include security requirements. Poor documentation also makes it difficult to understand assumptions made about the software, or why certain design choices were made.

Flaws may be present in the business or application logic, or in the system architecture. While flaws related to architecture, such as badly defined trust boundaries, may be difficult to identify without proper knowledge and an accurate overview of the system, flaws related to logic can be even harder to spot. Logic flaws encompass a range of vulnerabilities relating to privilege manipulation and transaction control manipulation [7]. A specific sequence of methods with specific argument values may lead to an unanticipated program state, which is difficult to find even with security testing.

Web-based applications have additional security factors to consider, such as data transport over unsecure networks, maintaining state, and whether to implement logic on the client side or server side. Du says that the higher number of security vulnerabilities in web applications may be due to the nature of the web [8]. Security measures were not built into the networks and frameworks used to access the internet. There are also no built-in methods to maintain state between a client and a server. The task of keeping data secure falls on developers.
Mobile applications have similar issues. Web and mobile use is skyrocketing; Pew Research Center reported in January 2014 that 58% of adults in the US own a smartphone [9]. Patton from HP Fortify states that "Many application teams have not been through core security training. During development of new applications, they often make incorrect assumptions that lead to security vulnerabilities. This seems to be particularly true of mobile development teams who are often rapidly building apps." [10]

Why is it Important to Know How to Find and Fix Flaws?

McConnell states in *Code Complete* that per 1000 lines of code (KLOC), the industry average is 15-50 errors [1, 11]. Even in software thoroughly tested for security vulnerabilities, there are an estimated 5 vulnerabilities per KLOC, according to McGraw. If the software is only tested for functionality, not security, that number is closer to 50 vulnerabilities per KLOC [1]. This has an enormous impact when the code base may be millions of lines of lines long.

This implies that the more lines of code in software, the more vulnerabilities it will have. In addition to a large codebase, complex systems also have more interactions, are harder to test, and are difficult to design [12]. Therefore, they are likely to have more vulnerabilities. The codebase size of some popular or well-known software in millions of lines of code were reported in 2013 [13]. See Table 1 for examples.

<table>
<thead>
<tr>
<th>Software</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mars Curiosity Rover</td>
<td>5 million</td>
</tr>
<tr>
<td>Boeing 787</td>
<td>6.5 million</td>
</tr>
<tr>
<td>Firefox Browser</td>
<td>9.7 million</td>
</tr>
<tr>
<td>Android</td>
<td>12 million</td>
</tr>
<tr>
<td>Windows 7</td>
<td>40 million (in 2009)</td>
</tr>
<tr>
<td>Facebook</td>
<td>62 million</td>
</tr>
<tr>
<td>Mac OS X 10.4</td>
<td>86 million</td>
</tr>
<tr>
<td>Healthcare.gov</td>
<td>500 million</td>
</tr>
</tbody>
</table>

Table 1: Codebase Sizes

McGraw estimates 50% of defects are flaws [2]. Using Windows 7 as an example, there could be an estimated 200,000 – 2 million security vulnerabilities, 100,000 – 1 million of which could be flaws. These software security issues have the potential to impact a huge number of individuals and businesses; Windows 7 controls just over half the global market share of desktop operating systems [14].

The average web application has 16.7 serious vulnerabilities, and "According to a recent report, over 80 percent of web applications have had at least one serious vulnerability." [8]. Vulnerabilities in code are inevitable. Being knowledgeable about logic flaws and following recommended security practices can at least significantly reduce the number of potential security issues. For software in development, security can be
addressed at the start. Existing code can be tested for flaws manually, and discovered flaws can be fixed.

**How to Reduce Flaws During Software Development**

When it comes to developing new software, security is more effective when intentionally built-in, instead of added to a completed product. It should also be addressed throughout the entire development cycle [2, 15, 16]. Developing software with security from the start could make a difference in the number of vulnerabilities by an order of magnitude (see the Windows 7 example in Table 1).

In general, assume any content accessible by the user can be manipulated by an attacker. This includes user input, data presented to the user or stored on the client side, parameter values, and the order that functions are called.

Several resources provide specific examples of logic flaw exploits and explain how they could be prevented. *Seven Business Logic Flaws That Put Your Website At Risk* [4] walks through seven real-world attacks to break down how the application was exploited and how it could have been prevented. *The Web Application Hacker's Handbook* [3] reviews eleven exploits and their prevention methods relating to poor input validation, circumvention of workflow, information leakage, and abuse of functionality. Table 2 shows the category of these flaws, some recommendations to prevent the flaws, and a summary of the examples.

<table>
<thead>
<tr>
<th>Category</th>
<th>Recommendations</th>
<th>Example Exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abuse of Functionality</td>
<td>Don’t display full usernames on a website [4]</td>
<td>Lock another user out of their account with multiple login attempts with incorrect password</td>
</tr>
<tr>
<td></td>
<td>Be careful when reusing code [3]</td>
<td>Same code for user and admin reset of password. Admin doesn’t need old password to reset.</td>
</tr>
<tr>
<td></td>
<td>Do not assume privilege level based on which function is used, or how a function is used; use the session for identity decisions [3]</td>
<td>Leave old password field blank on user password reset and it’s treated as an admin password reset</td>
</tr>
<tr>
<td>Circumvention of Workflow</td>
<td>Do not let audit trails be deleted, or be very careful about who has that ability [3]</td>
<td>Step 1: From admin account, create a second account and assign admin privileges. Step 2: From second account, delete audit trails from Step 1. Step 3: Perform malicious actions anonymously</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Information Leakage</td>
<td>Users can access steps in a multistage process out of order, or skip steps; the process should handle this properly [3]</td>
<td>Place online order, skip to delivery information page to avoid payment</td>
</tr>
<tr>
<td></td>
<td>Do not validate input on the client side when handling sensitive data, even if the data is encrypted [4]</td>
<td>Find hidden list of encrypted coupon codes on client side, brute force to break encryption and get free item</td>
</tr>
<tr>
<td>Be careful when reusing code [3]</td>
<td>When creating a new account or switching accounts on the same application, create a new session token instead of modifying data in an existing one [3]</td>
<td>Session token for online banking contained unique user ID and other personal information. When logged in, register account as another user. Session token information is modified, attacker can skip some access control steps since token was already authenticated, and access other users account information.</td>
</tr>
<tr>
<td></td>
<td>Bind the session id to the unique user ID on the server side, instead of placing the unique user ID on the client side [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If searching is allowed but the content is protected (must be paid for), assume any information that is displayed can be used [3]</td>
<td>Contents of search results are behind a paywall, but number of results found is still displayed. Use complex search terms to find if results exist. (Ex. “Company X”</td>
</tr>
<tr>
<td>Issue Type</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Insufficient Process Validation</td>
<td>Input should be verified both at initial content submission, and if that content is edited at a later time [4]</td>
<td>Comment screened at submission, but not when edited. Edited comment includes inappropriate text or malicious code.</td>
</tr>
<tr>
<td></td>
<td>When decisions are made based on specific criteria, verify the criteria at the final step of the process, in case it has changed [3]</td>
<td>Add items to shopping cart to receive discount. Remove items before paying, but discount remains.</td>
</tr>
<tr>
<td></td>
<td>Verify submission time against any input constraints [4]</td>
<td>Place order on stock exchange website at time A, but don't finalize it. At time B, check if price is now higher. Finalize submission, but pay price from time A.</td>
</tr>
<tr>
<td>Poor Input Validation</td>
<td>Users can add parameters and values that the function does not require; the function should handle this properly [3]</td>
<td>In multi-step online order process, a hidden cost parameter is set at a later step. User can modify cost parameter at an early step to get lower price</td>
</tr>
<tr>
<td></td>
<td>Verify parameter values are present, correct, and within allowed ranges [3]</td>
<td>Send negative bank transfer to avoid transfer limits</td>
</tr>
<tr>
<td></td>
<td>Be particularly careful when escaping</td>
<td>Nested characters can</td>
</tr>
</tbody>
</table>
Table 2: Recommended Flaw Prevention Methods based on Actual Exploits

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Recommended Action</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predictable Resource Location/Insufficient Authorization</td>
<td>Do not assume that hidden URLs cannot be discovered [4]</td>
<td>Hidden webpage is created for future news release and URL is based on the date. URL is easily guessed, information is used to buy or sell stocks ahead of the news release.</td>
</tr>
<tr>
<td>Weak Password Recovery Validation</td>
<td>Avoid secret questions with limited answers, or answers that could be discovered via publicly available information [4]</td>
<td>Secret question asking for favorite colors is easy to guess.</td>
</tr>
<tr>
<td></td>
<td>Password resets should involve an out-of-band method [4]</td>
<td>Secret question is easy to guess, attacker gains immediate access to account.</td>
</tr>
</tbody>
</table>

However, these are fairly specific recommendations. Some general best practices for preventing logic flaws during software development include the following:

Use detailed and thorough requirements, for both functionality and security [2]. This way security is addressed throughout development, and not just an afterthought. It is better to include security to prevent attacks, instead of reacting to a successful attack [2]. In later stages of development, tests may be based on the requirements [17], so security requirements may increase the amount of security-based testing. Tests also may be based on use cases, and some types of security testing can be based abuse and misuse cases [2], therefore define abuse and misuse cases in addition to use cases during the requirements phase [2, 18]. This also helps explain what the application shouldn’t do and how it should react to attacks or failures, and defines how the software should respond to abnormal input or logic flows. To develop the abuse and misuse cases, identify the business rules of the application [18], especially for rules relating to timing or sequences, monetary transactions, process workflow, human resource rules, and contractual rules. In the long run, it is cheaper to fix security issues at the start than after software has shipped [2, 19].

During development, the design of the application should be reviewed. Assumptions made by the application should be challenged to understand how it will handle unexpected user behavior and input. Also check the dependencies and interactions between different components. An architectural risk analysis [2] can help identify attack
resistance, ambiguity and weaknees in the design, attack surface area, and trust boundaries. Risks found from this analysis, or identified in other steps, should be tracked throughout the development process using a risk management framework [2]. A risk is defined as the probability of a vulnerability being exploited times the business impact.

Throughout development, use detailed and thorough documentation [2, 3]. The documentation should list assumptions made in the design. Code comments are a good way to document design choices, by explaining why a component was coded that way.

When testing the software, very assumptions that are made [3]. This includes the order of multistage functions, and when the state of the application is updated. It is especially difficult to maintain the correct state in a web application when there are more than two parties involved, such as the user, merchant, and payment collector, since only two parties can communicate at a time. All input should be verified [3] whether it is from the user or another server.

Include audit trails, and log everything [20]. Do not allow these to be deleted [3].

If the application has password-based access control, follow good password practices [19] such as safe storage of passwords, and strong password requirements.

If the application is web-based, let the server control the logic of an application, instead of the client [2, 7], since an attacker can’t bypass logic on the server side as easily.

**How to Test for Logic Flaws:**

Testing for flaws is complicated, because different types of logic require different types of tests. This is one reason why flaws are difficult to find manually and even harder to scan for using tools. Flaws are also more complicated to fix than bugs. Bugs may only exist on a single line, such as using the non-safe C method strcat(). The fix is as simple as using the C method strlcat() instead. Flaws often involve the design logic, and need to be addressed in multiple areas of the code [1, 2].

Logic flaws can be very subtle. Some creative thinking is needed when it comes to testing for logic flaws, whether you are testing an application under development or an existing application. Start by using the security-based requirements to develop test cases. Then use the design recommendations from the previous section as a guide for additional testing.

As stated earlier: assume any content accessible by the user can be manipulated by an attacker. This includes user input, data presented to the user or stored on the client side, parameter values, and the order that functions are called. For web applications, don’t assume input from another server is safe. Don’t assume input or data verification is done on another server. Even if another server currently verifies the value of something, that may change in the future.
Functional testing primarily checks if software features work properly under normal use, and does not typically include any security testing. It would not be used to find logic flaws, however, testing for logic flaws can be done alongside functional testing. Table 3 shows an example of this.

<table>
<thead>
<tr>
<th>Test Level</th>
<th>Description</th>
<th>Typical Use</th>
<th>Security Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component testing (aka Unit testing) [21, 22]</td>
<td>Test a single unit or component of the software that can be tested individually</td>
<td>Verify basic functionality of units such as objects, classes, or programs</td>
<td>Verify input exists, has correct values or is within allowed value ranges. Verify the function can handle unexpected input</td>
</tr>
<tr>
<td>Component Integration Testing (aka Integration Testing) [21, 22]</td>
<td>Tests the interaction and interfaces between components. May use a simulated environment.</td>
<td>Verifies functionality between components, or between a component and the operating system, the hardware, etc. There may be several levels of integration testing. The tests should be created systematically, such as top-down or bottom-up</td>
<td>Test how a value input in one component affects the interaction to a second component, or the program state after the value is processed by the second component. Test assumptions made by one component about a second component.</td>
</tr>
<tr>
<td>System Testing [21, 22]</td>
<td>Tests the whole integrated system. May use simulated external interfaces.</td>
<td>Verifies functionality when the software is installed in the environment.</td>
<td>Black-box testing, White-box testing. Test if the security preventions act differently in different environments, such as a web application on different browsers.</td>
</tr>
<tr>
<td>System Integration Testing [23]</td>
<td>Tests the system integrated with external interfaces</td>
<td>Verifies functionality between multiple systems</td>
<td>Black-box testing, White-box testing. Test assumptions made about the external interfaces.</td>
</tr>
</tbody>
</table>
Table 3: Logic Flaw Testing Alongside Traditional Testing

The Microsoft Developer Network provides some general information on testing, and breaks down the type of tests to perform when doing Black-Box or White-Box testing [24]. Black-Box is useful for testing edge cases of parameter values and creating a cause and effect graph. White-Box is useful for more complicated logic flaw testing, by doing complete path testing, branch/decision testing, condition testing, data flow testing, or loop testing.

*The Web Application Hacker’s Handbook* [3] and the Open Web Application Security Project (OWASP) [6] have recommendations for testing the logic in web applications. OWASP also provides links to tools that can aid in testing web applications. These can be broadly categorized as manipulation of data, and manipulation of workflow.

Assume all data sent from the client side, or another server, has the potential to be manipulated. It is important to test how the application reacts to incorrect, malicious, or missing data. In an HTTP request, “…try removing in turn each parameter submitted in requests, including cookies, query string fields, and items of POST data.” [3]. This includes removing both the value, and the parameter name itself. If the function being tested is part of a multistage process, continue through the entire process to check how that parameter affects the end result. If a function also is also used to generate values for users of different privilege levels, can a user with a lower privilege level still manually submit parameters that the user with the higher level privilege is supposed to submit?

In addition to the function parameters, check if a session token for one user can be updated or modified with information about a different user, since accounts can be accessed by different users on the same computer. And in general, check if sensitive data, like a unique user ID, is stored at all on the client side. [3].

If an application allows files to be uploaded, the application should be checked for how it responds to unexpected file types and malicious files [6].

When testing if the workflow can be manipulated, see if individual steps from a multistep process can be accessed directly. The steps order should also be tested by running a single step multiple times, skipping steps, and running steps out of order. For each of these tests, run to the end of the process if possible to see how the end result is affected, not just the intermediary steps. Overall, this is trying to test the assumptions made by each step of a multistep process, how those assumptions could be violated, and how the end result of the process is affected. This type of workflow manipulation can allow for privilege escalation, if access control is not verified at each step of the process. [3, 6]
OWASP also recommends testing the functionality of security defenses against application misuse [6].

The testing method used in the case study from How to Shop for Free Online [25] was Black-box testing for web applications, similar to the recommendations made by The Web Application Hacker's Handbook [3]. The authors did the following to find logic flaws in eCommerce software integrated with a third-party payment system:

When testing a process, start by manipulating the data. First, identify all of the arguments used in that process. If an argument is unsigned, test unexpected values. If an argument is signed, test substituting the same argument from a second session. For all arguments, test removing the argument completely. Manually trace through the entire process to see how each individual argument affects the internal state of the process. It is also possible for the user to manually add extra arguments to an HTTP request, so this should be tested as well.

Then, test manipulating the workflow. For multistep processes, try skipping one more steps, and running the steps out of order. The workflow manipulation should be done first with expected argument values, and then with manipulated data values from the previous step.

**Fixing Logic Flaws**

How to mitigate a logic flaw will depend on the category of logic, such as authentication or an eCommerce transaction, and what the logic flaw is. Authentication may have a fail-open flaw allowing an attacker to access an application even if authentication failed.

An example might be if in the authentication code, the session was set as authenticated before performing a SQL query to verify the username and password. With normal execution, the result of the SQL query is then evaluated, and an if-else statement verifies the username and password pair, and invalidates the session if the username or password is incorrect. However, if an abnormal SQL query was executed and it caused an exception, the if-else statement would be skipped and the attackers' session is still set as authenticated. To fix this, the exception must be handled, including invalidating the session.

For eCommerce transactions, an attacker could purchase something for free or at reduced prices, or use the same payment on multiple transactions, depending on the logic flaw. To fix these, additional verifications could be added to confirm the payment amount with the price of the order, or the payment status of a session could be cleared once an order has bene completed.

To help understand how to mitigate a logic flaw, find the assertions that should be made for the logic being used. For authentication logic, assert that a user should only be logged in if the username and password exist in the database, and they match. For eCommerce transactions, assert that the payment amount should equal the price of the
order, and the payment was intended for that order. What verifications are already in place for these assertions, and how were they bypassed by the flaw? What verifications could be added?

Why do Manual Testing Instead of Automation Testing?

“Automation of business logic abuse cases is not possible and remains a manual art relying on the skills of the tester and their knowledge of the complete business process and its rules.” -OWASP [6]

Automation testing requires defined patterns, and therefore works well to detect bugs, but application logic is inconsistent; one web application may implement the majority of authentication logic on the client side, while another places it on the server side. Testing for logic flaws also depends on the type of exploit [6]. Since there are no well-developed automated tools available to identify logic flaws, it must be done manually.

“Logic implies human thought required.” [7]. It is imperative to learn to how to identify and mitigate flaws in existing software, and use recommended practices during development.

Several groups do have prototypes to automate this process, however there are many limitations. Unlike bugs, flaws don’t have a common ‘signature’ to scan for [3]. Humans can identify behavioral patterns better than software tools, and human input is still needed for extracting the logical model of the application or verifying the results from current prototypes. Automation relies on patterns, but logic may be unique to an application, or to the domain/industry the app is used for. It is possible that tools could be developed where the custom business logic of an application would be input during setup, and then the tool would perform repeatable testing [7].

Pellegrino and Balzarotti [26] state that only general logic vulnerabilities may be found through existing automated tools and vulnerabilities specific to a particular application must still be found through manual inspection. They use an automated black-box approach by monitoring network traces when users interact with an application, and then extracting behavioral patterns. The goal was not to create an accurate model of the application, but to show a simple model of the application logic was sufficient to identify logic flaws during automated reasoning. Test cases were also generated based on the simple model. The prototype does not require the source code of the application, and is meant to be application-agnostic. It was successful at identifying flaws, but has several limitations. First, it is focused only on the classes of logic flaws found in eCommerce applications, and does not cover instances where the attacker is a malicious merchant, or the attacker can tamper with messages sent between the merchant and payment service. The final results from the tool still need manual review to distinguish between presentation issues, where the internal state is correct but displayed data is not, and cases where the internal state is not correct [26].
Felmetsger et al. [27] developed a prototype called Waler that monitors regular execution of a web application that uses Java servlets, and infers relationships between variables. The goal was to identify the apps specifications, without human feedback. Limitations include the need to monitor real users interacting with the application, though it may be possible to use a tool such as Selenium [27, 28] to automate user browsing activity. Waler relies on checks implemented in the applications' code; if the developer never checks that a number input by the user is negative when it should be positive, Waler assumes both positive and negative numbers are permissible. Currently, a third-party tool called Daikon [29] is used to derive the program specifications from the data gathered by Waler, so Waler is bound by the scalability issues of Daikon [27].

The case study from How to Shop for Free Online [25] verified their manual results with an automated tool called Corral [30]. A description of how Corral was used and its limitations are covered in the next section.

**The Case Study**

To help you understand the complexity of logic flaws and learn how to identify and fix logic flaws, here an example of a real-world application with an exploitable logic flaw. The case used here is the eCommerce software Interspire integrated with PayPal Express, presented in How to Shop for Free Online: Security Analysis of Cashier-as-a-Service Based Web Stores [25]. The exploit allows an attacker to use the payment meant for an inexpensive order on a completely different, more expensive order. This case study covers the logic involved in maintaining state between three parties, maintaining state in an eCommerce transaction, and enforcement of workflow.

Wang et al. [25] used a black-box method and manual code review to identify logic flaws in real eCommerce software, then verified them using an automation tool. The black-box method was to identify arguments in API calls that an attacker could potentially tamper with during the HTTP interactions between a user, merchant, and third-party cashier. Then, those modified arguments were traced through the entire transaction in a manual code review to identify how they affect the internal state.

The logic of the merchant software Interspire and the APIs it used for PayPal Standard, PayPal Express, Google Checkout, and Amazon Simple pay, was extracted, simplified, and implemented as functions in a program header file. A C program with symbolic execution used nondeterministic values to assign values to arguments and execute five random API calls available to the attacker, then call the final API method to complete the checkout process. A tool called Corral [30] loops through this second program multiple times, and verifies the assertion that the payment amount made to the third-party cashier was the same as the gross amount of the order.

The included PowerPoint identifies the API arguments in the Inerspire/PayPal Express case, and traces through code manually. It animates step-by-step the round-trip HTTP used by the attacker, the merchant, and the third-party cashier (PayPal). This
represents the public APIs provided by Interspire merchant software and PayPal Express. After each API call, the code representing the back-end software is shown with a table tracking the variables. The PowerPoint explains the flaw found by Wang et al. [25], where the attacker could use a session with a successful payment status on a different order than in was intended for. It allows the attacker to complete an expensive order using the payment intended for a cheaper order.

There will then be a set of discussion questions where you will be asked to identify and fix logic flaws in provided code, using this black-box method with manual code review.

Finally, you will run Corral with code provided by Wang et al. [25]. The code was specifically set up for use with Corral; it calls 5 methods in a nondeterministic order, with nondeterministic argument values, and then calls the final checkout method. Corral then loops through this code, though Wang et al. encountered an out of memory error if it was looped more than 6 times. This tool is not useful on a wide scale when testing an application with many API methods, and the logic of the application must be extracted into a separate program.

References


