The Art Of Exploiting Logical Flaws in Web Applications

Sumit Siddharth, *7Safe part of PA Consulting*
Richard Dean, *Portcullis Computer Security Ltd.*

Blackhat Briefings Abu Dhabi - December 2012

In last 5 or so years we have seen a rapid demand for web application security testing. Often security testers gets blinded by the traditional input validation flaws such as Cross Site Scripting or SQL Injection and can at times ignore the a critical part of the test which is assessing for logical flaws. Over the years we have identified some insane logical flaws and we have decided to recreate some of our best logical flaw hacks so that others can learn from these. Logical flaws are often difficult to find and anyone without an understanding of how the application is supposed to function would be oblivious to their existence and potential impact. This paper will takes you through some of the techniques and key ideas used when trying to identify logic flaws in applications as well give advice on how to minimise the inclusions of these problems in the first place.

Introduction

It is important to first establish what we mean by a logic flaw. In this instance we are referring to the behaviour of an application within the set of bounds defined by the original design. Moreover we are talking about the workflows through an application, you may be able to manipulate parameters to get a a SQL server to do something unusual or be able to inject code to get to your end game but it not these type of flaws that this paper will deal with. We are looking at the business logic of the application, so the control statements, If this then that etc. We are looking at corner and edge cases where the original designer had not properly accounted for what the user can do to manipulate the system.

Attack and Defence

As an attacker, the identification and exploitation of these logic flaws can be very fruitful and pleasing. Often many hours of hard graft need to be put in for little success, but then something so mind-numbingly stupid will emerge, that the effort seems worth it. Logic flaws, like with so many things, become easier to identify the more experience is gained. Not every application will be susceptible to them, but as an experienced tester you can fairly quickly get the feel of an application and so know that you are or are not likely to find any bugs. It is very difficult to explain what you need to look for but it is hoped that this paper and the associated presentation will highlight the key areas that should be looked at and what are the giveaways that constitutes the feel for an experienced tester. Likewise from the development point of view what can be done to minimise the likelihood that your applications contain these type of bugs. If you are protecting against SQL injection you use parametrised queries if you are protecting against XSS you use a layer like OWASPS ESAPI, but for logic flaws there are not quick wins. You need to have clearly defined what the application needs to do.

Key Axiom

"You cannot comprehensively test for logic flaws unless you know what the application is supposed to be doing"

If you have an application test, unless you understand what it does and how a normal user would interact with it, then it is very difficult to identify what you are trying to look for in terms of logic flaws. You can probably very successfully run an automated tool across the site and pick up most of the parameter manipulation attacks, but you are very unlikely to find any of the hidden logic flaw gems.

In a simple case you may have a banking application with many different users, where each user should be able to see their details but not those of any other user. You can set up an automated crawler to check all of the functionality and accessible pages of user a, then do the same for user b. You may even then be able to automatically find all of those pages that are different for the users and check to make sure that they cannot be replayed as the other user. This way you are trying to find anything that is not explicitly allowed for user a by forced browsing to those requests from user b. Now without understanding the details of the site you may be able to identify a link that was not in user a's profile but they can see it. Now, have you found a problem? Unless you know the context of the request you cannot tell. For instance, it may that the application uses targeted marketing to supply generally available resources to customers that are likely to need them. There is no business reason why user a should not be able to see the link, just that it was not deemed they would like to see it. There are many much more complicated examples where a bug may be a feature and vice versa, that without having a human eye with an understanding of the business logic and security controls would have not been identified.

Inspiration from Other Areas

Whilst putting together some examples to illustrate logic flaws it was noticed that there were a number of problems found that were quite similar to vulnerabilities found elsewhere in other types of code. In web applications it is very common to see an authentication system ask for a selection of characters from your password. This is done so that you never expose the whole of your password to anyone that can record these details. Two interesting by products of this are firstly that you can infer that this section of your secret data is not saved in a hashed format in the applications database and secondly it may introduce a user enumeration vector. In a good system the user should be asked the same characters from their password until they get it right, it should not change each time you get it wrong. So if the app only asks for a username before you hit this first stage of authentication you can infer legitimate usernames. If you go to the same username twice and you get the same password indices twice you have a real account, if it changes you don't. In order to get around this problem we advise either have some form a secret before you get to the indices question, or that if the account is not real you create the indices from username supplied. This way, if real you read from a database so it is the same each time. If not real you calculate from the data given so it will be the same.

In bad implementations of this that we have seen the developers have taken the quick route and do not store this indices information in a database and sometimes do not even store it in the session. In the worst case, the client told the server which part of the password it was supplying which the server trusted. It was found that it was possible to supply index values which were past the end of the stored password therefore were null. So when the server checked the supplied value which was null against the saved version which was also null it allowed access.

This type of attack seemed very similar to the mysql authentication bypass bug represented by CVE-2004-0627[1]. In this the client would tell the server how long the password that it was sending in the authentication packet was. From the original advisory "The final loop compares each character in the 'scrambled' string against the string that mysql knows is the correct response, until there are no more characters in 'scrambled'. Since there are no characters *at all* in 'scrambled', the function returns '0' immediately, allowing the user to authenticate with a zero-length string'.' In both of these cases the server is trusting the user to supply some information which it trusts which forms part for the bounds for the authentication check. The fist getting the server to check a value larger than the password length with returns known value and the second forcing the server to not do any checks at all. What can we learn from this? Well as always you should not be trusting the data supplied by the user. Both of the

[1] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CAN-2004-0627

bugs occur because the server is changing a character at a time and neither performed a sanity check on the bounds before starting the operation.

Variable Reuse

Another analogy which we saw was related to a 2006 Black Hat talk by Halvar Flake, "Attacks on uninitialized local variable"[2] and also to a number of Kernel bugs I've seen. This talk discusses the use of variables in a function before they have been initialised and how this may lead to a security vulnerability in the code. In the Kernel vulnerabilities case the problem existed because the kernel checked that a pointer was valid but the user could change it before it acted upon it. This is a fairly common class of bug called TOCTOU (Time of check, time of use). When we started to discuss this topic an application that we tested number of years ago came up, but at that point we had not joined the dots. The bug was in a very complicated application that without an intimate understanding of what the app was supposed to do, or actually what the app was not supposed to do we would not have spotted.

An example, put into a context that can be explained without having to have working knowledge of the industry sector the original problem was found in, has been used to illustrate the problem. In both this bug, Halvar's paper and the Kernel bugs variables were acted upon as if they were in a known good state when in fact the user had been able to control them. In the web application the error that the developer has made is that there was a naming overlap between two session variables used in two sections of the application. It was possible to use one set of application functions to control a session variable then get a second function, which had already checked that the session variable, to act upon it afterwards. In this example it allows the messaging functionality of the application to alter a session variable which is used by the user profile editing page to decide which profile to edit. If done at the right time in the work flow this allowed a low privilege user to recover the details of all other application users and even take control leveraging the forgotten password functionality. Whilst the bugs context is different to the real world find, the way it can be detected is the same. Whilst analysing the site it was noticed that two significantly different areas of the application where using the same parameter names when submitting data to the server. With a leap of imagination about variable name resue a useful exploitation path was found a serious flaw in the application was uncovered.

These are just two examples that have come from the app side independently but put onto previously know issues in other places. The important question is are there more, are there classes of bug that have yet to be coded into applications? With AJAX and other asynchronous techniques used very widely now there may well be more timing problems that in the previous world a synchronous sites would never have existed. Whilst we cannot speculate on the weird and wonderful things that may be out there we can look at where they are likely to exist if they do at all.

Cracks In the Application

In the previous section we discussed a bug that was detected due to a similarity across different sections of a site. This is fairly uncommon, it is a lot more common to find logic flaws in applications where things are disjointed. These 'cracks' often exists where development has happened in different phases then brought together. This maybe because two sets of devs have been working on the application, or maybe because it has been developed at different times. These cracks, if they can be detected, often create bugs because code isn't quite compatible or someone doesn't quite understand how it should be integrated. There was recently (November 2011) a Facebook problem which appeared to exists for this kind of reason. Facebook for a long time had a (fairly) strong access model for users images. Unless allowed you should not be able to see other people pictures. Then at some point it was decided that there should be the ability to report abuse about a user. This abuse maybe the result of a user placing obscene images in their profile, if this was the case then the reporter should be able to let the Facebook Team, know which image was causing the problem. The vulnerability that was introduced is that in allowing for this abuse process to work as described, they also allowed everyone to see everyone else's images. This functionality completely bypassed the access control on the images if you wanted to report it as an abusive picture. Whether this was a design or implementation problem I do not know, but the end result was that image access control was now could be bypassed.

Once a crack is found there maybe may problems

[2] http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf

there to exploit, but finding the cracks is the hard part. So what are the things that you should look for? Conversely as a developer what are the things that you should avoid doing?

As an attacker I am looking to try to identify the joins an application, this is a gut feeling quite a lot of the time but this stems from experience. The sort of things that I look for are where obviously different coding styles have been used, indicators are things like the naming conventions of parameters, the way elements are set out on a page, maybe even HTML comments. You can often notice subtle changes in the way certain work flows behave when trying to circumvent them. You may have one section that handles errors well, where in another you get verbose messages back. Not only is this an indicator that input handling ids not good, but also that there maybe a crack which may lead to a logic flaw. Once a suspected crack has been found you now need to look at each side of it and see what functionality straddles the crack. You may have a report generation system on one side, and a secure repository on reports on the other. Can you manipulate the creation functionality to return more than just the report that you have just generated? You may need to get ID's and other pieces of information from one side and use them in the other. If it were one seamless app then probably not, but if it feels like a bolt-on then there is every chance that some weird combination of integration techniques done by the dev team may leave things open.

In order to try to combat these as developers you should be making sure that you have a good SDL and that when code is combined it is done so in a way that they new code interacts nicely with the old. Ideally you should be developing on a frame work. This will mean that any new code is compatible, and can be integrated smoothly with the application base. All new pieces should have gone through both security and functional testing. In the example of Facebook this problem would have been picked up a good security control document had existed for the current functionality. This should have detailed that none 'Friends' should not be able to see images and under statements like this the functional and security testing should have been done. Without this documented control a functional tester may not have detected the problem as they may be testing the functionality in isolation. Also security testing may not have picked it up an automated tool would have seen this as legitimate functionality. Only by knowing the overall stance on this from the application would you know that an issue even existed. Again back to the point you can't properly test unless you know what is supposed to happen.

Know Your Tech

Another crack that seems to emerge quite regularly is when developers do not fully understand the technologies that they are using. When integrating technologies such as external authentication or Windows file systems problems can occur as these have unknown features. In a test that we performed recently we found that the development Team allowed admins to control the location of the database backups. We'd already bypassed a large number of access controls to get this far in, and actually uncovered functionality and data in the application which indicated that the supplier had been lying to us. But thats a whole other story. The database backup could also be initiated from the web interface and was saved to a location of our choice. Because this was a windows system we could pass it a UNC path. So we set up a Samba listener and set off the backup passing it our IP address as the UNC path and down came the complete internal database. Unfortunately we could not use this connection to re attack the system but a copy of the database include all of the user accounts and passwords was good enough.

If you are going to introduce tech because you need a small amount if its functionality make sure there is nothing hidden in what you expose. Similarity make sure you are integrating technologies that you fully understand. In an application we saw recently the development Team had been told that they needed to use salted hashes. They then went away and used salted hashes in their authentication mechanism. Unfortunately they had not understood why they had been told to use salts and implemented a system where they had in effect made the hash stored in the database the authentication token rather than the password. This meant that an attacker did not need to know the original password to authenticate they just merely needed to know the hash.

Conclusions

Hopefully you'll appreciate the full circle we have know come, if you know Windows well, you'll know how it's authentication access and so be able to see the analogy here with pass the hash techniques which have been prevalent for over a decade[3]. Overall I'm

[3] http://corelabs.coresecurity.com/index.php?action=view&type=page&name=Modifying_Windows_NT_Logon_Credentials

a keen believer of automating things that can be automated but knowing what cannot be. At this point I do not think that we can do a good job of automating the process of trying to find these types of bugs. You need a good set of eyes to spot when something is illogical, even if the systems meets the functional design spec perfectly it may breach other security controls documented else where. As a developer you need to think carefully about the full implications of introducing new functionality, it may seem useful in the small sphere you are currently working in, but may be bad for the bigger picture. As a security tester you need to make sure that you fully understand the bigger picture even if just testing small sections of the application. As an attacker I often find it very useful to ask "What if?". Think up a scenario then go an investigate it. I should not be able to x, now if I were a developer how may I have written the code for this sections of the app and what may I have done wrong to allow me to do x. How can I verify this hypothesis. It's a scientific approach but I find that it works.

There are four key things that I want to emphasise:

1. You cant build or test an application properly unless you know what it is supposed to do.
2. Logic flaws very often exist in application cracks, learn how to spot these or how to make sure they are not created as port of a development process.
3. Make sure you have a good SDL, integrating security and continually testing at key stages not just at the end
4. Make sure you understand the technologies that you are using, or look for technology integration as these ofter introduce cracks