web-attacks.md

Web attacks: a brief recap

This document contains a brief recap on the types of attacks explained in the course.

SQL Injection

Let's examine the main kind of techniques used to compromise SQL databases.

1. **Quote balancing and commenting** to bypass further checks in a WHERE clause. Example:
   - Query: `SELECT * FROM users WHERE name = '$name' AND pwd='$pwd'`
   - If the PHP code simply take name variable from the `$_GET` variable directly, we can login as any user by simply inserting `admin' -- ` (trailing space matters). In this way, the check on the password is commented out and the login is bypassed.
2. **Tautology**: making a WHERE clause always evaluating to TRUE, thus bypassing it. Example:
   - Query: `SELECT * FROM users WHERE pwd='$pwd' AND name = '$name'`
   - Again, if name variable is taken directly from the `$_GET` variable, attackers can inject `admin' OR 1=1 -- `. In this way, the condition on password is evaluated in AND with the name admin, and the results of that is ORed with 1=1. Since this is always true, a contant true in OR with whatever condition is always true. So the each record in the database will match with this WHERE clause and the query will return all the users and the attacker can bypass authentication.
3. Attacking INSERT INTO and UPDATE to modify/update the content of the database. For instance you could register a user with more privileges, of course this is application dependent.
4. **Stealing information with the UNION operator**. This is really very useful but only if the result of the query are shown in some way, otherwise we cannot gain anything. The order of operations is understanding the number of columns of the query by using `UNION SELECT(NULL)`, then `UNION SELECT(NULL, NULL)` and so on. This is because of course the UNION operatore requires the same number of columns. Say the number of columns is three. So we can try to understand which of those columns are strings. Strings are useful because almost any kind of data can be casted to it. So you can try `UNION SELECT('a', NULL, NULL)`. If no errors are showned, column one is a string. Repeat on all the columns. Now that we know this, we can try to steal information on which tables are present in the database, using its metadata. For instance, on MySQL you can union with `information_schema.columns` to learn tables and their columns. Once you know that, you can retrieve any kind of info in the database. Do the SELECT of course on the columns that corresponds to strings, previously discovered. ORDER BY operator can also be used to know the number of columns the query returned.
5. **Inference** for normal blind SQL injection (query do not return any results). Induce the application to a detectable behaviour: for instance login successful or not. Example:
   - Query: `SELECT * FROM users WHERE name = '$name' AND pwd='$pwd'`. We would like to know the password of the administrator (supposed it is in clear, unrealistic I know).
   - Insert into name variable this payload: `admin' AND SUBSTRING(pwd, 1, 1) = 'a' --`. The final comment is used to comment out the condition and on the password. After injecting this, you observe whether you actually logged in or not. If you logged in, then for sure the password of the admin begins with letter a. Iterate over it and you're done.
6. **Piggybacked queries**. Unrealistic but they are the most powerful: you can do whatever you want, just paying attention to close the legitimate query, insert a semi colon and type it in.
7. **Time-based SQL injections**. Totally blind injection, no detectable behaviour in the app? Induce the query to SLEEP only if a condition is met. Then observe the response time. For instance: `SELECT IF((subquery), SLEEP(tot), NULL)`. This query will sleep if the condition in the subquery is true.

Countermeasures.

The best counter measures are of course the prepared statements. The idea is to build the query in two phases: first phase the code, second its parameter. In this way, all the user controlled input are treated as parameter and we can execute the query with no harm.

XXE (XML External Entities)

In the XML standard, markup strings preceded by "&" or ";" are called entities. The most common entities in XML are the one used for special characters, for instance &amp will be replaced by an ampersand. Entities can also be external: in this case, the value of the entity is retrieved dynamically by the XML parser. Take `<!DOCTYPE bar [<!ENTITY foo SYSTEM "http://example.com/index.html">]>` . This entity will induce the XML parser to perform an HTTP request to example.com. XXE can take place for instance when the application accept AJAX request. A malicious user can inject the definition of a new entity and be able to:

- Read files local to the server. `<!DOCTYPE bar [<!ENTITY foo SYSTEM "file:///etc/passwd">]>` can be used to retrieve the content of /etc/passwd.
- Server Side Request Forgery (SSRF). We can induce the server to perform requests to internal services that would not be reachable from the outside. This could help retrieving important data or more.
- Internal port and host scanning.
- Denial of Service: letting the server read a file with an infinite stream, such as /dev/urandom.

XSS (Cross Site Scripting)

This kind of vulnerability has the user as target. The aim of this attack is to let the browser execute a malicious script, typically a javascript script. Typically, what the javascript script does is stealing the session ID of the user, so that the attacker can impersonate him and access to its private space in the web application. There are many types of XSS attacks:

- Reflected XSS: suppose that a server has a page that gets an input parameter for displaying an error message. For instance www.example.com/show_error.php?msg=error+msg. The PHP will simply take into a variable the message and echo it on the generated page. An attacker try to let a client to follow a link like this one: www.example.com/show_error.php?msg=\<script>alert(1)</script>. The web server will produce a web page containing that malicious script and it gets executed on the client side, sending for instance the session ID to the attacker.
- Stored XSS: suppose the attacker can inject in some database the malicious script. For instance this could be saved as the content of a blog post. Then, every client visiting that blog post will receive a web page containing that script, and it gets executed. This is more powerful than reflected XSS because the user does not have to click on suspicious links and he is already authenticated since he is visiting the web site.
- DOM-Based XSS: suppose the web page contains a javascript script that checks the URL and performs a document.write(). Since the URL is controlled by the attacker, he can again insert a script in the URL and let the user follow the link, exactly the same way of the reflected XSS.

So, in general, payloads of XSS are:

- Session hijacking
- Injecting trojan functionalities: display a form asking for credentials or credit card informations. The form will appears from the legitimate site, this is potentially very riskful for users.
- If Stored XSS, we can try to let the attack self-replicate, creating a kind of worm.