

HTTP Headers

- HTTP headers pass between the client and server with each HTTP request and response
- While most web applications pay no attention to HTTP headers, some do and it is important to realise that as with all data coming from a client, HTTP headers are susceptible to manipulation
- Most browsers do not allow direct HTTP header modification but writing a program to submit handcrafted HTTP headers is straightforward
- Java, Ruby and Perl's LWP modules make writing such a program very simple and there are always web proxies such as WebScarab



HTTP Headers

- The `Referer` field is included by most browsers in the HTTP header and contains the URL of the web page from which the request originated
- Some web applications check this field in the belief it can be used to verify that a HTTP request is coming from a trusted source
- This is a mistaken belief as all HTTP header fields can be modified
- Similarly, other fields such as `Accept-Language` and `Agent` present in the HTTP header cannot be relied upon to contain trustworthy data



URL Encoding

- Only a subset of the ASCII character set can appear unencoded in a URL: `[0-9a-zA-Z], "$-_.+!*()"`, along with some reserved characters
- However an unrestricted character set is available to the web application. . .
- Other characters can be inserted in a URL using URL encoding: this involves taking a character's 8-bit hex representation and prefixing it with a "%" character e.g. `"%20"` encodes a space



URL Encoding

- Thus certain "unexpected" characters may be encoded and sent to a web application e.g. nulls or metacharacters of special significance to an underlying subsystem (e.g a database or operating system)
- As a result, essentially any data can be disguised in a URL and passed to a web application
- A web application cannot trust any data coming from the client and everything must be validated
- Suppose we do not adequately validate client-side data? What might happen? Your web application could be left susceptible to SQL injection or cross-site scripting attacks. . .



Injection

- We have looked at various types and formats of client-side data
- Often an attacker will seek to take advantage of a lack of server-side validation to inject code or data to cause malicious side-effects
- Usually an attack can be analysed as if composed of three components: prefix + payload + suffix
- The prefix and suffix allow the attack code to be spliced into the server-side code



SQL Injection

- Many web applications store persistent server-side data in a database and use SQL statements to interact with that database

- Consider the following server-side code:

```
command = ``select * from client where  
          name = ` ` + name + ` ` ``
```

- With benign input we build commands like the following:

```
select * from client where name = `Sonia`
```

- However, what happens if our user's name is:

```
`Sonia' or 1 = 1 -- ``
```



SQL Injection

- The resulting SQL command string is (where -- turns what follows into a comment):

```
select * from client where  
      name = `Sonia' or 1 = 1 -- `
```

- This statement returns all rows for which:

```
name = Sonia or 1 = 1
```

- With the latter true for all rows this user gets to see every entry in the database
- Attackers can potentially use this technique to create new entries and to update or delete existing ones
- Here the prefix is `Sonia``, the payload `or 1 = 1` and the suffix `--`



The Problem

- Requests sent to subsystems by web applications consist of two components: instructions and data
- The data part is often sent by the client
- If that data part contains metacharacters i.e. characters of special significance to the underlying subsystem then that data may cause the subsystem's command parser to "context switch" from data processing to instruction processing mode
- If that happens, data sent from a client may be interpreted as instructions by the subsystem
- Clearly our web application is now in trouble since client-supplied instructions can delete information, steal passwords, alter user privileges etc.



Solving The Problem

- In the examples so far the problem character has been the single quote: '
- It causes the SQL parser to exit string context and to begin looking for instructions
- We need to tell the SQL parser to treat the the single quote as just that and not as a metacharacter
- Taking away a metacharacter's special powers is called "escaping the character" and is normally achieved by prefixing it with an "escape character"



Solving The Problem

- A stored procedure is a subroutine stored in the database that can be invoked by a web application
- As such, stored procedures can help define a "database API" for your application and may help constrain the damage caused by a successful SQL injection attack
- Stored procedures also limit the potential damage when source code is compromised
- However, although valuable in their own right, stored procedures do not solve the problem of SQL injection
- The same can be said of applying the principle of least privilege and connecting to the database with only those privileges that are necessary for the correct functioning of the web application



Solving The Problem

- To solve the problem, we must ensure that any and all metacharacters are neutralised before being passed to any subsystem
- Writing your own "washing" function that escapes metacharacters is one approach: look up the subsystem's documentation and make sure to escape all relevant metacharacters
- The single quote is an obvious one, it can often be escaped by duplication e.g. replace ' with '' in all string parameters
- However, there may be other metacharacters: adopting this approach it is up to you to make sure you have accounted for them all



Solving The Problem

- Can we ask the database itself for help in handling metacharacters?
- Yes!
- SQL statements should never be built through raw string concatenation
- Instead we should use prepared statements wherein query parameters are passed separately to the SQL commands themselves
- With prepared statements, placeholders are used to ensure parameters function only as parameters and not as additional code (metacharacters are thus rendered harmless):

```
command = select * from client where name = ?  
command->execute($name)
```



Solving The Problem

- Practise “defence-in-depth”
- Distinguish between metacharacter handling and input validation or your application may be vulnerable to “second order injection problems”
- Validate all incoming data in the “input validation layer”
- Handle metacharacters when passing data to a subsystem (through sanitisation or, where available, by using prepared statements)



Solving The Problem

- Use stored procedures
- Implement the “principle of least privilege”
- Return uninformative error messages or an attacker may use them to map out database organisation to aid in an attack



Cross-site Scripting Attacks

- Like most web application attacks, XSS attacks are made possible by a lack of adequate data validation
- The subsystem being attacked is not server-side however but is the client-side browser itself
- A successful XSS attack can enable an attacker to cause the execution of arbitrary script during a user’s session with a trusted domain
- Crucially, the code runs in the context of that domain and can monitor and control the session, changing web page contents and sending information to the attacker e.g. session IDs, credit card numbers etc.



How Do XSS Attacks Work?

- Web servers often generate pages dynamically before sending them to a user
- If an attacker can control some of that content and it is not validated before being sent to a browser then she may be able to launch an XSS attack against another user
- As a simple example, a user enters “apple” in a search dialog box on some web site
- The web server responds with a page that says that the string “apple” could not be found



How Do XSS Attacks Work?

- User input has been used in the production of a page sent back to the user: we can thus potentially control the contents of a page produced by the server
- What if we searched for the following string:
`<SCRIPT>Alert (``Hello'')</SCRIPT>?`
- If this string were to appear in the search results a dialog box would be displayed containing the word “Hello”: we are delivering script to the browser
- In this case the browser is our own but it need not be...



Attack Examples

- The simplest kind of “attack” can be mounted against guest books that carry out no output validation
- Posting some JavaScript that redirects client browsers to “inappropriate” sites or downloads “inappropriate” images will not reflect well on the victim or the web site



Attack Examples

- XSS attacks can be used to implement session hijacking
- If a user must log in to view the guest book and an attacker has embedded within it the required JavaScript, the victim’s session cookie may end up in an attacker’s hands
- When the page is downloaded the client’s browser executes the JavaScript, it runs in the context of the domain that set the cookie and thus the script has access to it
- The script can post cookie contents to the attacker’s web site and session hijacking is straightforward from there



Attack Examples

- Using social engineering an attacker can target particular users rather than rely on them stumbling across her guest book entry
- A web site displays a form, a user fills it in and is presented with the form again if some fields are invalid
- Assume the filtering of form data is inadequate and script can be sent to the user’s browser
- An attacker can generate a link which submits URL encoded script to the server and send the link in an e-mail to a victim saying “Look at this offer!”
- The victim clicks on the link and the form appears but arriving with it is the script encoded in the link; this script has been bounced off the server and into his browser (a reflection attack)



Attack Examples

- The link looks trustworthy but the script executed does not originate with the trusted domain
- The script however, having been delivered by that domain, runs in the security context of that domain
- It can therefore modify web page contents, access cookies and send information back to an attacker
- Note that the script need not even appear in the link, a link to the script will do and that link may itself be URL encoded to further disguise its presence
- These kinds of vulnerabilities are extremely common with new ones announced regularly



Solving The XSS Problem

- XSS attacks are another incarnation of the metacharacter problem
- Thus to solve the problem we must take away their special meaning from HTML metacharacters
- Escaping HTML metacharacters is called “HTML Encoding”
- With HTML encoding every & is replaced by &, ' ' by ", < by < and > by >
- When should HTML encoding be applied?



Solving The XSS Problem

- HTML encoding should be applied whenever the application generates some output destined for the client's browser
- There are three reasons why filtering should be delayed until output time. . .
 1. It is not just user generated input that need be HTML encoded, all data retrieved from external sources should be encoded and the “encode before sending” rule is easiest to remember
 2. Storing HTML encoded input in a database may cause problems for an application that wishes to subsequently process that data - it will have to be unencoded
 3. Storing HTML encoded strings requires more room than unencoded versions



Cross-site Request Forgery

- An attacker may be able to cause a victim to carry out unintended web requests simply by sending them an e-mail or pointing them at a web page
- This is a CSRF and it shows that an attacker may not need to hijack the session of an authenticated user in order to carry out some malicious action
- Instead they exploit the fact that if a user follows a link to a web server to which they are already authenticated then their authenticating credentials (e.g. a cookie) will be automatically submitted along with the request
- The attacker tricks the user into carrying out the malicious action themselves
- These kinds of attack are also known as Web trojans as the trap is concealed in something innocuous like a link



Cross-site Request Forgery

- Consider a form submitted by a GET request: to have a victim submit the form an attacker need only send them a link and hope they follow it
- The victim does not know that following the link has possibly serious side-effects
- Even if the form in question is submitted by a POST request a link will still do the job if it leads to a web page containing script that auto-POSTs the form to the server
- An attacker could even e-mail the auto-POSTing form and hope the victim is using an e-mail client that automatically handles HTML e-mails (and the script they contain)
- In this case simply reading the e-mail will cause the form to be submitted



Cross-site Request Forgery

- What harm can submitting a form do? Quite a bit!
- If you are logged into and have been authenticated by your online bank then most likely you have a session cookie identifying who you are to the web application
- The online bank may allow you to transfer money to other accounts by means of a form
- The attacker may then be able to have you view a web page that auto-POSTs form details that transfers money to her account, not good!
- The script does not run in the security context of the target domain and cannot access its cookies; however, by directing the victim's browser to the target domain the script indirectly (i.e. without ever seeing them) exploits security-sensitive cookies to enable the attack



Cross-site Request Forgery

- The problem is that users are submitting to the server forms that were never presented to them
- Checking the `Referer` header would solve the problem in many cases but there is no guarantee it will always be available to the web server
- Online banks take the approach that all critical actions require reauthentication and this does solve the problem but at the expense of usability
- Can the problem be solved without requiring regular user reauthentication?



Cross-site Request Forgery

- One solution involves keeping track of the forms that have been presented to each client
- With each form sent to a client's browser, send along a random number (in a hidden field perhaps) and also store a copy of the number in the server-side session
- Whenever a form is submitted verify the "ticket number" it contains matches the server-side session copy
- Any form submitted by an attacker on a victim's behalf will not contain a valid ticket number and will be rejected server-side



Session Fixation

- Web applications use session management to provide a convenient user experience to client
- Session IDs are an attractive target for attackers and once obtained permit session hijacking
- For this reason web servers protect against: interception, prediction and brute-force attacks
- A fourth class of attack against sessions is session fixation
- *Session Fixation Vulnerability in Web-based Applications* by Mitja Kolsek



Session Fixation

- In a session fixation attack the attacker fixes the user's session before they have logged into the application and thus the need to obtain the session ID afterwards is eliminated
- The user's session ID is fixed in advance instead of having been randomly generated at login time
- The attack proceeds differently depending on whether the session ID is distributed by URL, hidden field or cookie
- The attack works when a session ID is not regenerated after logging in
- The attack works even when session IDs are distributed over HTTPS



Session Fixation

- The attacker visits the web site and is assigned a session a URL based session ID:
`https://www.bank.com/login?sessionID=1234`
- The attacker then takes this link and emails it to the victim along with a message to say the bank has collapsed
- The user follows the link, the web server sees the sessionID is valid and presents the login form to the victim
- The victim logs in and is taken to
`https://www.bank.com/accounts?sessionID=1234`
- The attacker has the session ID because crucially it did not change across the login procedure



Session Fixation

- Note the login procedure is not necessary: the victim may go on to make a purchase or enter some personal details that could be viewed by the attacker
- If the session ID is kept in a cookie then the attacker has some more work to do
- One way is to implement a XSS attack: the user is sent a link and by following it some script is bounced via the server to their browser
- The script runs in the security domain of the server and creates a cookie with the desired session ID
`document.cookie="sessionID=1234";`



Validation Strategies

- Many common attacks can be prevented by appropriate data validation
- Both input and output should be validated
- Three common approaches to the problem:
 1. Accept only known valid data (whitelisting)
 2. Reject known bad data (blacklisting)
 3. Sanitise bad data
- By far the best strategy and simplest one to enforce is the first although there may be occasions where one of the other two are required

Validation Strategies

- Rejecting bad data is risky as it is difficult to maintain an up-to-date list of web application attack signatures, dangerous metacharacters vary according to subsystem etc.
- Sanitising bad data is similarly risky
- Data validation involves checking:
 - Data type
 - Syntax
 - Length
- Regular expressions, for which Perl has extensive support, are ideal for performing this kind of task and are your first line of defence