

IIT Madras
ONLINE DEGREE

Modern Application Development – 1
Professor Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Security Mechanisms

Hello everyone, and welcome to this course on modern application development.

(Refer Slide Time: 00:17)

**Security
Mechanisms**

For the Web

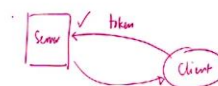


Now, that we have seen different forms of access control, let us look at the security mechanisms that can be used to implement this in the context of the web.

(Refer Slide Time: 00:28)

Types of security checks

- Obscurity (generally very bad idea):
 - application listens on non-standard port known only to specific people
- Address:
 - where are you coming from? host based access/deny controls
- Login:
 - username/password provided to each person needing access
- Tokens:
 - access tokens that are difficult/impossible to duplicate
 - can be used for machine-to-machine authentication without passwords



So, the first thing is, what are different kinds of security checks that can be applied in order to control access to a given application. And one of them, sort of the most trivial and also a pretty bad idea on the base is what is called security by obscurity. And what obscurity refers to is something that is obscure, is something which not many people know about. That is sort of the definition of obscure. And what I mean by that is, let us say that I take a regular HTTP application, and instead of running it on port number 80, I then run it on port no 1356, something which is unknown except to the people to whom I sort of give the information out.

So, it listens on a non-standard board that information is given out only to specific people, and therefore I think that only those who know that port, will be able to access the application.

Now, this is generally a very bad idea. There are multiple ways of getting around this. One of course it could be as simple as let us say one of the people to whom I send it, then accidentally just leaves a page open or somebody walks past their desk and happens to see this thing open and sees the port number. And then they go and just sort of forward that information to a whole bunch of other people and before you know it, everybody knows that there is this particular port out here.

Now, this might sound very silly but the point is it has been done in the past. So, do not sort of dismiss it and say nobody will ever do this. There have been cases where things of this sort have been done. Please make sure that you do not think of such ideas. You could put a further restriction which says the server itself will also check which IP address you are coming from. And this is what is called host-based access control. And essentially what it is saying is that the server has access to the IP address from which the request came, and you might sort of whitelist certain IPs and say only if you are coming from a particular IP address you will be allowed to come in.

Now, this is in fact done in a lot of cases. Maybe not so much for web applications, but for example SSH access, remote shell access to machines, sometimes what you say is I will allow you to access this particular machine only if you are logging in from some other known IP address, which means that you first need to be able to connect to that machine and from there connect to this new machine.

And by sort of showing that you were able to connect to that first machine to start with, you have in some sense proven to the system that you have enough permissions to be able to get into this new machine. It has its own problems. It can be used in certain circumstances but the biggest problem over here is that what if that intermediate machine somehow ran into problems, and essentially was broken into. Then anybody who has access to that now suddenly has access to your secure application or your secure machine. So, this by itself is usually not considered a good enough form of enforcing security.

Now, login mechanisms are much better because they are sort of enclosed at the final server level. Whichever server you are trying to access will itself put up a query asking you, saying, look you need to give me a username and password. Now, how are username and password is implemented? We will look at a couple of examples of those later but there are a number of different things that come up over here as well and it is, the full discussion of that is more or less beyond the scope of what we can do here, including things like how should you be storing passwords, what should you not do with usernames and passwords.

In particular one of the things is the server should never store the passwords that it receives directly in text. Why? Because at some point the server might get broken into by someone who should not have access to it, at which point they go, look over there and find that all the passwords of all the people who have logged in there are available, directly as text. And there is a good chance that a person might use the same password here as they do somewhere else, and what that means is it is not only have your server been broken into, it has now opened up a whole lot of other servers also to potential attack.

There are ways of getting around this. Usually what you do is you store one kind of a so-called hash of the password, which cannot be reversed but it is uniquely identifiable. Like I said, it is sort of beyond the scope of what we can do here. For those who are interested you can read up more about how security mechanisms are implemented.

Now, apart from a username and login, there might be circumstances where, let us say that I want my laptop, or my server machine to directly be able to connect to some other

server somewhere, and get some information. I need to be able to show that I am a valid user, but at the same time I cannot really sit over there and type in usernames, passwords, click on captchas, various other kind of things because it is just the machine or the script in the background which is supposed to do this connection.

In such cases usually what is done is, you have the final server and you have the client out here, the client will give some kind of a token which can be recognized by the server and the token then basically says okay, accept, and allows you to pass that information back. So, only if the token is valid will the server respond to the client and give the information back as requested. This is especially useful for machine to machine authentication. It sort of replaces the idea of a password. It combines username and password into one token. The whole point is that the token should not be possible, immune to guess. It is usually some long string.

It can be arbitrarily long because after all these are just machines, they are just sending bytes one after to the other. They do not need to remember anything. That information is going to be stored somewhere either in memory or on the disk. So, this kind of token can be used especially in these contexts. Of course, you need to be careful to make sure that the token is not exposed, nobody else can sort of see what is there in the token because then they could use it. So, these are all different kind of security checks that can be implemented in the server check.

(Refer Slide Time: 07:09)



HTTP authentication

Basic HTTP auth:

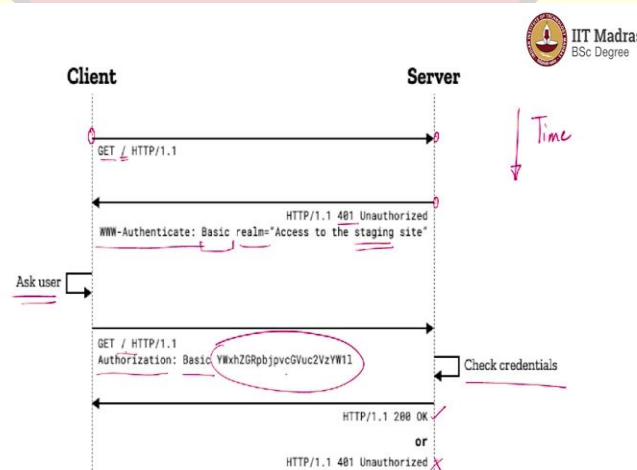
- Enforced by server
- Server returns "401/Unauthorized" code to client
- Contrast with:
 - "404" - not found
 - "403" - forbidden (no option to authenticate)
- Client must respond with access token as an extra "Header" in next request

Now, in the context of HTTP, there are a few mechanisms that are available, which we are going to look at now. The first is what is called the basic HTTP authentication. And this is part of the HTTP specification itself. It is enforced by the server. And you might have come across this in certain places where essentially what you get is that the server returns the so-called 401 code. So, the 401 is a specific HTTP status code which indicates unauthorized. The client is not authorized to get this particular piece of information.

So, the all the four codes in some sense are some kind of HTTP level status errors. The most, the one that you are probably most familiar with is the 404, the not found. That just basically says that, you ask the client for the, client ask the server for something, the server could not find what you are asking for. So, it is giving a backup message saying I do not know what you are looking at, looking for.

There is also something called a 403, which is an outright forbidden. It just says you are not allowed to access this. 401 is a little bit more nuanced than that. It is not saying it is outright forbidden, it is saying right now, given what we have, the information I have right now, you are not authorized, this client is not authorized to access. So, how did should the client authorize themselves? And typically, that is done by responding with some kind of an access token.

(Refer Slide Time: 08:50)



Src: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication>

Now, this is an example of the sort of back and forth that would happen and it is from the again the Mozilla developer network. What happens over here is, of course, the first

thing, the client initiates the conversation. It gives a get request to the server and the server responds. So, this is essentially an access of time, basically showing here, as time progresses what are the requests that are sent back and forth.

So, as a client, I try to request the /url. The server basically responds with a 401 unauthorized, and in addition, it also gives me a WWW-Authenticate header, which gives some information about the type of authentication that I am willing to accept, and it quickly says, this is basic authentication, and this realm is a sort of identifier which says where you should go look for a password. So, realms are just basically used in order to give some more clarity about what kind of login, the server is expected to see. And the reason you might have this is, I might have multiple realms associated with a given site.

So, let us say that I am a user who is basically connecting to a particular website, I might find that there is one realm associated with the admin dashboard, another realm associated with the user space, and they might even be completely different from each other. That is probably an unnecessary way of doing it, but it is possible, I could do something of that sort.

Or as in this case it might say that there is a particular realm associated with the staging site, which means that the user names and passwords that you use for connecting to a staging site which is some temporary testing scenario is different than what you would use in production. So, that is why the realm is important.

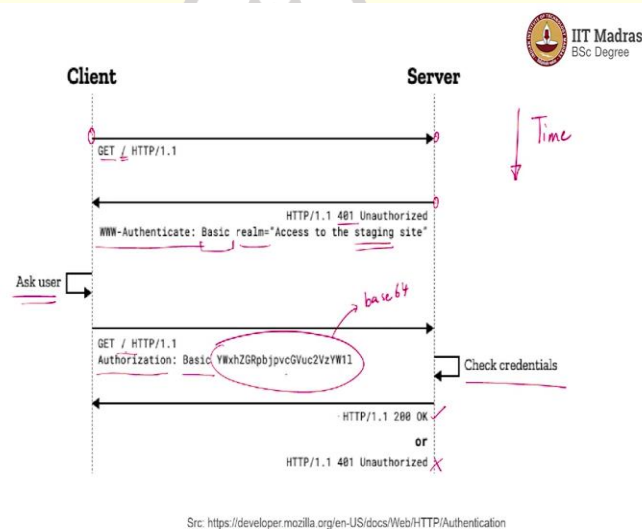
So, once the client has got the realm, it will ask the user, one pop-up window will come over there and say enter your username and password. It will then send back this, once again a get request, but now it has on x bar header. Authorization, basic. And this is some code.

How is the code generated? It basically takes the username, the password, combines them together and does something called base64 encoding. This puts the two together and converts it into a different format which basically looks like this. Goes to the server, the server will check the credentials, whatever is encoded over here, it can decode, and find out what was actually sent. And if it worked, it will say okay. If it did not accept that username and password it will basically say unauthorized, and you can try once more.

(Refer Slide Time: 11:45)

Problems with HTTP Basic Auth

- Username, Password effectively sent as plain text (base64 encoding)
 - Some minimal security if HTTPS is used (wiretap is difficult)
- Password will be seen in cleartext at server
 - Should not be needed - better mechanisms possible
- No standard process for "logout"



So, the mechanism itself looks straightforward. What are the problems? The first thing is that, like I said, this thing over here is essentially uses something called base64 encoding, which is very easy to reverse. So, if I take a username and password and put it through base64 encoding, then I can just do a base64 decoding and get back the username and password. So, anybody who sees this token basically knows your username and password.

Now, of course what that means is that I need to make sure that nobody can get that information. Only the client and the server are allowed to see that. And you will get to

that problem ultimately a little bit later, when we are talking about wire level security. So, some minimal security is possible with HTTP is used, but the problem is that the password will still be seen as clear text. Clear text meaning, the actual password that you typed in, abcd1234, whatever it is that you type, will be visible to the server.

Now, you might think obviously the server needs to check if the password is correct, but that is a problem because it means that the server now has the password. It is sitting somewhere in the server's memory. It might potentially get written into disk and the server itself had the raw password sitting somewhere in it so that it could check. How else can you do this? Like I said, the one-way functions, the hash functions that can be, there are things that can be used in order to convert the password into something else, which the server actually does not need to store at all.

And finally, one other problem with HTTP Basic Authentication is there is no standardized process for logging out. Strangely enough, I mean they have this thing where it, as long as you keep giving that token, authorization token, it basically says you are logged in you can continue. But there should be some way of explicitly logging out and saying I do not want to continue this connection.

Let us say that I want to go away, I am done with this, I was using it in some public kiosk. I cannot log out. One way is usually to sort of shut down the browser and so on, but it is unfortunately not standardized. So, it is possible that some browser might still retain your username and password.

(Refer Slide Time: 14:07)



Digest authentication

- Message digest: cryptographic function
 - eg. MD5, SHA1, SHA256 etc.
- One-way function:
 - $f(A) = B$
 - Easy to compute B given A
 - Very difficult (near impossible) to compute A given B
- Can define such one-way functions on strings
 - String \rightarrow binary number

$f^{-1}(B)$

A slightly better variant of HTTP Basic Authentication is what is called Digest authentication. Now, in Digest, what happens is that, like I said, all of this now starts coming into the realm of cryptography. And cryptography, essentially it deals with how to store and transfer secrets. Crypto essentially stands for secret. So, cryptography deals with how you can use mathematical functions, and mathematical property to say that certain kinds of operations are going to be very difficult to perform, and therefore can be the basis of securely transferring data from A to B.

Now, there is, there is something called a one-way function. And the idea of a one-way function mathematically is very simple. It basically says that, I have some mathematical function which takes in a number A, and it generates another number B. I am saying number, but remember that it could even be a string. An ASCII string at the end of the day, is just a set of bits which means that I can think of it as a number, if necessary.

So, I could think of that A as either a string or a hexadecimal value or a binary value. Whichever way it is, at the end of the day it is just basically a number. And $f(A)$ is a function which takes one number and converts it into another. Alternatively, I can also say it takes one string and converts it into another string.

Now, what are the properties of a one-way function? The forward function is easy to compute. That is, if I am given A, $f(A)$ is easy to compute. On the other hand, it is nearly

impossible to do the reverse, that is to say, if I give you B, can you get back A such that $f(A) = B$?

Normally, we would call this f^{-1} . And this is nearly impossible to compute. There are ways to prove this mathematically, for certain kinds of functions. So, those functions are chosen as the so called one-way functions, and Digest authentication essentially does some kind of complex set of operations with such functions.

(Refer Slide Time: 16:34)

HTTP Digest authentication

- Server provides a "nonce" to prevent spoofing
- Client must create a secret value including nonce
- Example:
 - $HA1 = MD5(\text{username}:\text{realm}:\text{password})$
 - $HA2 = MD5(\text{method}:\text{URI})$
 - $\text{response} = MD5(HA1:\text{nonce}:HA2)$
- Server and client know all parameters above, so both will compute same
- Any third party snooping will see only final response
 - cannot extract original values (username, password, nonce etc)
 - nonce only used once to prevent replay

The way that it does this, it also uses a concept of something called a nonce, which is basically a number used once. And what that means is the server has some kind of internal random number generation of its own, and whenever the client tries to connect, it generates a nonce and sends it back to the client. The client must now prove that it has certain information needed by creating a new secret value which includes this nonce.

And an example of that would be that you take the username, the realm, password, and pass it through MD5, which is one of the functions. It is a one-way function, that we discussed earlier. Create another HA2, which is again an MD5 function of some other string. Finally take HA1, the nonce, HA2, add them all one after the other and create an MD5.

Why are you doing all this? Just to make it that much harder for a person who is trying to break this system to be able to invert all of these functions. Even if they somehow

manage to invert one MD5 function, which is as it is quite difficult to do, they would now need to do this multiple times in order to get back and get all the way down to the password which ultimately is what we need.

So, the server and the client on the other hand know all of these parameters, which means that they can compute the same secret value on both sides. And the client has shown to the server that it was able to compute the same value that the server did. Any third party who snoops on these values, even if they see the response will not be able to make anything useful out of it.

And because it is a nonce, it is a number used only once, it means that each and every time you are trying to access you will re-compute this. So, even if a person somehow manages to get hold of one of these tokens, they cannot invert the functions and get your password and they cannot use this same value again because the nonce would have changed. So, Digest authentication is slightly better in battery than basic authentication.

(Refer Slide Time: 18:50)

Client certificates

- Cryptographically secure certificates provided to each client
- Client does handshake with server to exchange information, prove knowledge
- Keep cert secure on client end
 - Impossible to reverse and find the key



There is yet another form of authentication which is called a Client certificate. Now, most of you are probably familiar with HTTPS. We will be talking about that again shortly, but HTTPS, at least you know that these are secure websites. Secure in what sense? Secure in the sense that there is some certificate that the server is showing you, and providing to you, which you have reason to believe is authentic. We will get to that in another part of this lecture.

Now, in the same way I could also do the other thing. I could say that a client needs to show some information to the server to convince the server that it is an authentic client. Of course normally we do that by using passwords. What we are saying is you could use the same kind of mechanism, that server certificate, but now do it on the client's side, and the client now is able to establish to the server that it has some information which only that particular client could have known. It has a specific certificate which was issued by somebody that both the server and the client trust, and therefore this server can happily trust this client.

Now, like I said, this is just yet another of the mechanisms that can be used. Ultimately all of these things have very similar kinds of problems. You need to make sure that the client certificate is kept secure. Nobody should be able to go in and just steal the laptop and get the client certificate off from it. If that happens then, you have essentially lost the security of the thing. So, physical security might be as big a problem as anything else in this context.

(Refer Slide Time: 20:42)

Form input

- Username, Password entered into form
- Transmitted over link to server
 - link must be kept secure (HTTPS)
- GET requests:
 - URL encoded data: very insecure, open to spoofing
- POST requests:
 - form multipart data: slightly more secure
 - still needs secure link to avoid data leakage

? username = abc & password = xyz

Now, what happens when we are using an HTTP form? So, in a form input, usually the username and password are entered into the form and are transmitted over the link to the server. So, the name and password pretty much have to be sent as plain text. Now, at least the one thing that you can try and do over there is, you try and secure the link between client and server so that nobody can sort of tap into it and directly see the

password, but you also need to make sure that the password is provided in a reasonably secure manner. In particular, if you are using get requests, what happens with get, there is no concept of a request body. You only have the url encoded data.

So, after the url you will basically have some question mark and let us say username equal to something and password equal to something else. And if I have a string like this, this is a very bad idea because requests that are sent like this are very often logged. Even the server itself will directly take the request and store it in its log files. In between if there is some kind of a proxy server that might also take it and store it into log files, which basically means that the password is now sitting out there in the log file as plain text. And you do not know who has access to those files.

So, get requests are generally very bad idea for forms that have user names and passwords. Do not use them. But post requests are better that way. At least they do not get logged. And you can use it in order to send this information. If you combine a post request with a secure link which is established by HTTPS, that is pretty much as good as you can do at this point.

(Refer Slide Time: 22:38)

Request level security

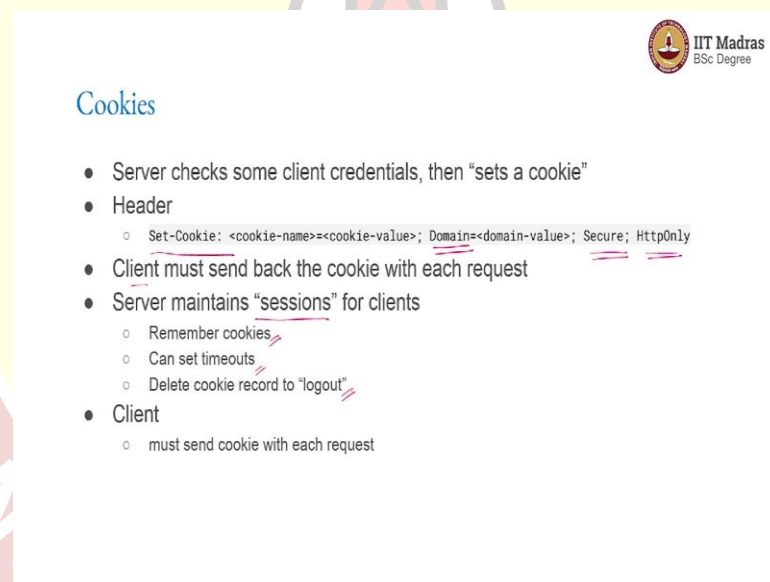
- One TCP connection
 - One security check may be sufficient
 - other network level issues to consider for TCP security
- Without connection KeepAlive:
 - each request needs new TCP connection
 - each request needs new authentication

Now, each request that is going from the client to the server requires a certain degree of security. And one way of doing it is I have one server, one client, the client establishes a connection to the server, if I need to have it in such a way that for each and every request

I need to make a new connection, then for each and every connection I need to re-establish my authentication.

On the other hand, if I can somehow keep that connection alive and send multiple requests across it, that is good news because as far as the server is concerned it is not possible to break into a connection between it. That is sort of guaranteed by the network level itself. It says that okay, unless you are involved in setting up the connection from the beginning, it is not possible for a third party to come in and suddenly start giving new information out here. So, if you can do a single connection, keep it alive, and have multiple requests going across it, that at least reduces the overhead required. It does not change your security requirements, it just sort of reduces the number of times you need to do this.

(Refer Slide Time: 23:47)



Cookies

- Server checks some client credentials, then "sets a cookie"
- Header
 - Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>; Secure; HttpOnly
- Client must send back the cookie with each request
- Server maintains "sessions" for clients
 - Remember cookies
 - Can set timeouts
 - Delete cookie record to "logout"
- Client
 - must send cookie with each request

Now, all of these were mechanisms that are available with the core web itself. There is one more mechanism that is available which is commonly used and which we will talk about in a little bit more detail later. For now, I just want to kind of mention the idea. And this again is a term that pretty much every day, I am sure, you are coming across some page or the other where it says this site uses Cookies, click ok to accept. So, what is a cookie? A cookie is just once again, a random number or a random piece of information which the server sends back to a client. It might be after having checked whether the client is allowed to authenticate, or it might just be something for tracking.

It might be okay, I am sending this to the client, and the next time this client comes back to see me, the client should give me the same key that helps me to say oh okay wait I saw you last Thursday. That is where you left off, now, what shall we do now. So, a cookie can just remember that the HTTP protocol was stateless, or it is stateless, meaning that the server does not know what the client, client's present state is. A cookie is a way of getting around that. By sending a cookie along with each request the client is sort of constantly updating the server and the server can then change the cookie, send it back and the next time it gets back this cookie it knows exactly what was the last part of the conversation it had with the client.

So, this cookie can be used in order to create a sense of state. It is a piece of memory that is shared between the server and the client, and that is exactly what state means. The idea is that the server can basically use this header, set-cookie, in order to set some kind of information. There are some additional parameters such as which domain is it valid for, should it be used only on an HTTPS link and so on. This finally also says it should not be used with JavaScript requests, only with HTTPS, something of that.

So, there are a few sort of extra parameters you can set in cookies. But the main thing is that a client must take whatever cookie it has received from the server, and send it back along with each request as part of the header information. Now, by looking at these cookies, the server can maintain sessions for clients. It basically says okay, I have got this cookie from the client. I last saw that client so, such and such time ago, this is what they were doing, this is the page they were looking at, send back something corresponding to that.

Next time a request comes, once again the cookie is there, the server has this information that this is what it sent last to the client, and this is how we can update it. Now, the server then, for doing that it needs to remember all the cookies that are sent. It can set timeouts, it can also just delete a cookie at the local end and say look, this client is logged out. Start from the beginning again.

Now, that gives a lot of power both to the server and the client. On the server side what it means is that it can enforce login, log out, it can timeout people and disconnect them if

necessary. On the client side you can just refuse to send an equal case, which is what is the so-called incognito modes, or the private browsing modes and browsers.

At least they do not send the usual cookies that were there. They might still accept cookies, in certain cases, but then they are just going to delete them after the session is over, which means that every time you open a new private browsing window, it sort of does not send cookies again to the server.

The server thinks this is a new client that is coming along, establishes a new connection, and after all that when you close the window all the cookies are deleted, they are not resent. So, it gives some amount of power both to the clients and to the servers.

(Refer Slide Time: 27:50)

API security

- Cookies etc. require interactive use (browser)
- Basic auth pop-up window

APIs:

- Typically accessed by machine clients or other applications
- Command-line etc. possible
- Use “token” or “API key” for access
 - Subject to same restrictions: HTTPS, not part of URL etc.

Now, finally a word on the API level security. So, APIs are also things where, as we saw in the example of CoWin, booking appointments and so on, you might want to have authentication or some form of security. Now, cookies typically require some kind of interactive use, they need a browser it can be done in some cases with the command line as well, but it is a bit harder to do that. And in particular if you come back to the 401 code and you need to pop up a window for authorization, an automatic client is not going to be able to do that.

So, for access to APIs, typically servers also provide another mechanism which is related to something called an API key. So, what is a key? I basically register with the website.

Maybe the GitHub, or the people who are organising that, and they issue a token to me. And what that token means is, I need to keep that token safe, I should not share it with anyone else, but on every request that I send to the server I just need to attach this API key header and send the token along. The server will look at that and say, okay I know that I issued this key to such and such a user, they are allowed to access these parts of the system, give them back information.

Good news, of course. In fact, the server can also invalidate tokens, it can set a timeout on them, lots of things. It gives the server a five degree of control. And as far as the client is concerned, if somebody accidentally stole your token, or you exposed it somewhere, you can go to the server and delete the token and say, look, do not accept this anymore. Issue a fresh token.

