# Session Authentication: An Overview

Ian Burnett, Comp 490, Fall 2022

## General Info

Security - an all-important, ever-evolving field. I'm not an expert, and this is not professional advice, but this paper should offer a good springboard for your own research.

This in mind, there's one takeaway that I'd like to impart to everyone reading this - security is never *secure*. If there's a lock meant to be opened by one person, someone else can figure out a way to open it, too. The goal of security is not to make an attack impossible, but to make it so difficult that attackers won't find it worthwhile to attack *you*. The methods discussed here are not infallible, and repeatedly in my research I found that protocols often described in terms of "defeating" attacks had vulnerabilities listed in other articles. Again, nothing is *secure*.

## What is Authentication?

Authentication is the process by which a user of some software is identified and permissions are granted to them. This can be accomplished in a number of ways, but the underlying principle is for a user/client to establish a secret with the server that will be shared with no one else. In doing so, if an unidentified client presents the server with this secret known only to the server and a registered client, the server "knows" this unidentified client is actually that particular registered client, since no one else possesses that secret - seems reasonable enough!

The most common authentication protocol is the use of a password, so our client Jan registers a (public) username: *Jan* and a (private) password: *dragon* with our server that retrieves information from a legal record database. Every time Jan wants to search for all active warrants in their state, they present their credentials (*Jan* and *dragon*) to the server, which looks up their public identifier *Jan* and checks if the private/secret identifier *dragon* matches. Since they do, it accepts that it must be Jan making the request, and returns the warrant list.

Now, what about Lilith, a malicious actor who shouldn't have access to the legal database? She may be able to find that Jan's public identifier is *Jan*, since it isn't kept secret, but she can't make a valid request without guessing Jan's private password *dragon*. This keeps the legal database safe by restricting access to only users who register a private password, a process managed by the database owner.

## What is a Session?

In the above authentication example, Jan has to provide their credentials with every request, which can get tiresome very quickly. To remedy this, we need to create some way of remembering Jan so they can provide their credentials once, at the start of any information-retrieving session.

Wait, there's that word! Session. So, it comes about pretty naturally - a session in security terms is referring to a persistent authentication that lasts for the duration of what we'd call a "session" in plain English.

Again, there are several ways of accomplishing this, but the most common way is to ask for credentials at a "log-in" portal, then return a randomly generated session "token" - a secret similar to Jan's password that the server can return to Jan's client once they get authenticated using their credentials. This token, say *JanValid42*, is then stored in Jan's client and automatically sent to the server with every request. Since it's known only to Jan's client and the server, and is issued only upon Jan supplying valid credentials, the token is a sort of "proof" of a valid authentication, so the server can use it to authenticate requests coming from Jan without them having to supply their credentials every time.

Thus, Jan can log-in once, their client receives the token, and the session is established for as long as the server is set to accept that token. Jan can "log out" to wipe that token from their client and tell the server to no longer accept it, or the server can invalidate the token on its own after a set expiration period. Since this method still requires a valid password to retrieve a session token, Lilith is left in the same position of having to guess a secret password to gain access.

If things were *just* this simple, however, security wouldn't be such a large field. So let's tackle our first issue with authentication - password secrecy.

## Password Secrecy

You may have noticed a glaring problem in the above explanation - the password is assumed to be secret, but what if it isn't? Ridiculous as it may seem, what if Jan shares their password with Lilith because they ask nicely? Then Lilith can pretend to be Jan, and access the database whenever she wants.

This is called social engineering, and is one of the hardest attacks to defend against. People are messy and forgetful, they have priorities beyond security (like family), and they sometimes trust blindly. They may write down their credentials in an obvious place (tantamount to storing a key next to its lock), hand over their credentials to a perfect stranger who pretends to be a family member asking for help, or even to someone with a clipboard who looks official and asks. You can't eliminate these risks so long as humans are part of your system, but implementing organizational policies about software security can help mitigate them. We won't cover social engineering in depth, but it's an ever-present threat that should be kept in mind during all discussions of security.

A much more addressable concern for a programmer is "what if Jan's password gets stolen from the database? What about during transit?" Vulnerabilities happen, user info gets leaked from databases, users log-in over unsecured connections that get intercepted, and so-on. What can be done?

## *HTTPS & HSTS*

First, let's address insecure transmission. There are an incredible number of ways to intercept internet traffic, but for the sake of simplicity, we won't be worrying much about the particular methods used, just their capabilities.

So, what if when Jan logs into the server to retrieve some legal documents, Lilith is set up to eavesdrop on their traffic, sees Jan send *Jan* and *dragon* to the server, and sees the server reply with the new session token *JanValid68*? Now, Lilith has both Jan's credentials, *and* the token which says valid credentials have been supplied, offering two avenues of attack.

The first option is the ideal - Lilith can use the stolen credentials to create a new, separate token and behave as normal through the official client. So long as this went unnoticed, this is perpetual access. However, for reasons that will become clear later in the paper, gaining credentials is difficult - it's far more likely to only be able to steal the token.

In *that* scenario, Lilith could "hijack" Jan's current session by writing her own client program to interact with the server. After all, a server is just a program with a web address that accepts certain commands predefined by its code, and a client is just a program that provides a convenient user interface for sending commands to the server. If the commands are known, *anyone* can write a client for a server. All Lilith's client needs to do is bypass the log-in and just insert Jan's stolen token with every request, which the server would accept as proof that these requests were coming from Jan's authenticated session. This access would end eventually when the token expires, but Lilith could steal a lot of information, or even try to use this temporary access to update Jan's credentials to something only *she* knew to and gain perpetual access.

How can we fix this? Traffic is normally conducted through HTTP - Hypertext Transfer Protocol, which is a hugely important concept that's beyond the scope of this paper. A sufficient understanding is that HTTP messages come in requests (from client to server) and responses (server to client), and are composed broadly of a method (which specifies generally what the message is trying to do), path (what web address the message is being sent to), header (which contains meta-information about the message), and a body (which contains additional data required to complete the request). These requests are sent in plain text, just like you're reading here, so Lilith could read Jan's and the server's messages without trouble.

However, there is HTTPS - Hypertext Transfer Protocol *Secure*, which is a similar protocol using the same message format, but conducts all the transmissions in an encrypted fashion. Essentially, the server has a certificate (vouched for by a trusted authority, called a CA/Certificate Authority) which allows it and the client to establish a truly secret key pair through a [TLS handshake], allowing client and server to encrypt and decrypt each other's messages.

So long as the private key associated with the certificate held by the server is kept secret (which is done through very strict protocols, and if a breach of privacy is discovered, a new one will be issued), all traffic conducted through HTTPS will be securely encrypted. Not invulnerable, but there is no known way for a third party to *read* the messages without the key pair that's been securely established using the server's certificate.

So, if our server for the legal database implements HTTPS by obtaining one of these SSL (Secure Socket Layer) certificates from a trusted CA, Jan can transmit their credentials securely, and Lilith might be able to read the traffic, but the contents will be encrypted gibberish. However, Jan might get lazy, forget to type the *s* in the web address to the client app, and connect over plain HTTP - now Lilith can read the messages again!

That's where HSTS (HTTP Strict Transport Security) can help. HSTS is a server setting that attaches a header (a field containing meta-information about the message) telling clients that the traffic *should* be conducted in HTTPS to every HTTP response, and upgrades the connection if it wasn't already HTTPS. This helps mitigate any damage caused by user negligence in initiating a secure connection, but doesn't fully solve it - if Lilith is intercepting Jan's connection in a way that lets her stop their messages and alter them before they pass to the server (and vice-versa), Lilith can prevent the HSTS header from being returned to Jan. This only works if Jan's connection is HTTP, and thus only on Jan's first[1} connection to the server, but that one time is still a real threat that we can't fully solve ourselves.

There is some discussion about having every site that implements HSTS noted in DNS (Domain Name System) servers, which handle routing to websites when you click a link/type a URL. This would allow them to mandate an HTTPS connection without an HSTS header already getting returned, patching up that vulnerability. That said, sometimes even DNS servers are compromised (DNS spoofing), further illustrating the point that nothing is guaranteed in security. One other consideration is closing the ability to connect to HTTP entirely[2] for the server which hosts a website that's intended only for HTTPS connections. This does seem to solve the problem on the surface, but is usually avoided because of the reality of how websites are used. The most important issue is that closing HTTP access means most browsers would simply display a "bad connection" page rather than redirecting to the HTTPS address, and most users would consequently give up on the endeavor assuming the website was "broken". While security is important, it's only an issue because websites are meant to be *used* - after all, the most secure website would be one that isn't actually on the internet at all.

---

[1]*Technically, most HSTS headers carry an expiration date, allowing another HTTP connection after that point. Thus, it's not only the first, but rather* whenever *a single HTTP connection is allowed before an HSTS header is returned.*

[2]*This means closing port 80 on the server - if you don't know what a port is, here's a decent overview.*

## *Hashing*

Now that Jan can transmit data securely, except for personal blunders, how do we mitigate the damage done when[3] leaks *do* happen? Well, we make the data hard to read! We can't establish a one-time key pair for every user account without storing one of said keys in the database[4}, so instead of encryption, we turn to "hashing".

Hashing, like encryption, creates a hard-to-read string of seeming gibberish - unlike encryption, though, hashing doesn't have a way to *reverse* that process. Hashing is (practically) a *one-way function*[5]. That might not seem very useful, but when you add in the other major property of hash functions - that the same input for the same hash function will always produce the same output - it becomes *very* useful.

Consider - Jan's password needs to be a shared secret with the server, but that doesn't mean they both need to see the exact same secret. If every time Jan enters their password to a log-in/sign up portal, it gets hashed to look like *8621ffdbc5698829397d97767ac13db3*[6] to the server, and no other input than *dragon* will produce that hash, the server can authenticate Jan using this unique hash just the same as if it received *dragon*[7]. What's the advantage, then? Well, the password *dragon* and the hash of *dragon* can be decoupled, with the database only ever storing the hash, and Jan only ever storing *dragon*. And since hashes can't be reversed, how useful is Lilith going to find that hash if she finds a flaw in the server that returns Jan's user info? Not very, since she can't know that the input that produces the hash is *dragon*, and passing the hash value of dragon into the hashing algorithm will produce a completely *different* hash[8] than just passing the original *dragon*.

If our log-in/sign-up system hashes passwords as they're entered, so the database entry and lookup process deals entirely with hashes, Lilith has a lot less to work with even though Jan's secret is still just *dragon*.

However, as with everything, hashing has pitfalls, so let's take a look at the arms race that's happened there (thus far).

## *Cracking Hashes*

So, if hashes are one-way functions that can't be reversed, how in the world is Lilith going to cause problems for Jan? In a word - guessing. After all, you don't want to use a simple password because a friend who knows you well might be able to guess it - an attacker with a powerful processor might not make guesses with high accuracy, but can make them with a high *quantity*.

3 *Never treat security as an if, always as a when.*

4 *See prior comment about storing keys with locks.*

5 *As a very brief explanation, they rely on large prime numbers and their factors, then dividing them and swapping the remainders, which results in a shared number that no one has figured out how to reverse-engineer algorithmically.*

6 *For those wondering, this is* dragon *passed through the MD5 hashing function.*

7 *Further details of transmission will be covered in a later section.*

8 *Passed through MD5 again, it becomes* dcdd71b83858fcaebef3b7a266247af8

Lilith can come up with some parameters to programmatically generate a list of possible passwords to guess, run them through the hashing algorithm our software is using (like MD5), and then compare Jan's leaked hash against this generated list of paired passwords and hashes that were guessed. If Jan's hash matches one of the hashes generated from a guess, Lilith can look up what guess *produced* that hash, and learn that Jan's password is *dragon*, since that's the input that generated the matching hash. Now, that's a bit cumbersome to do - generating a new list of potentially *millions* of hashes every time you want to crack a hash can take a lot of time, so why bother generating it each time?

Lilith can just generate a few of these lists for different parameters and/or hashing algorithms *once*, then every time she wants to crack a hash, she just has to run the comparison check between the leaked hash and each relevant pre-generated list. Hashing can take a long time for a computer compared to a matching check, which can be performed on the order of hundreds of millions per second, depending on processor power.

If Lilith is checking for every possible password hash in a particular range (say from 1-7 alphanumeric characters), this is a brute force attack, and while it will exhaustively crack all the passwords in that range, the list gets exponentially larger. For upper and lowercase letters, plus numbers and "special characters", that number[9] is 95 raised to the power of however many characters long of a password is being searched for. For a 7 character password, that would be $95^7$, or about 69 *trillion* possible passwords. Even at 100 million hashes per second, that would take 690 thousand seconds (about 8 days) to generate, let alone search. Many well designed passwords are *much* longer than that, so brute forcing often falls flat.

The better way to go about it is to run a "dictionary" attack - rather than generating (or borrowing) a list of every possible permutation, Lilith can draw that list from a set of already known passwords that have been cracked in the past. After all, people tend to reuse passwords, and use similar passwords to each other. For example, *dragon* typically falls in the top 50, alongside *password*, *password123*, various common words and expletives, and so on. Lilith can bet good money that you yourself use some pretty similar passwords, if not the *exact* same one in multiple places. Now, with this dictionary list, Lilith would hit 100% of those top 50 from the linked list, rather than the 14%[10] that are 7 characters exactly that she could brute-force in about 8 days. Obviously, the numbers are inflated because not everyone uses these passwords, and these passwords disproportionately are 7 characters, but the point of efficiency stands - Lilith's matches are going to return a lot more results for the same amount of time by

[9]*That's 26 upper + 26 lower + 33 "special" + 10 numerals.*

[10]*This becomes 60% if you include 7 characters* or less. *Every 6 character password would take only about 2 hours at this rate, so is reasonable to consider.*

working a bit smarter and leveraging information that people openly share[11].

Using these dictionary attacks, Lilith can reasonably make short work of cracking any passwords she wants so long as they've already been cracked or stored improperly elsewhere. And, by programmatically trying a few variations (such as generating hashes of the entire dictionary, but with common replacements like *a* to @) she can find quite a few passwords that *haven't* been fully cracked yet. As it stands, that sounds like the time and effort to just look up a hash's corresponding password is almost as easy as reading it in plain text, so what's been done to combat this?

## *Salting & CPU Loading Hashes*

If Lilith is using pre-generated lists of hashes, the most obvious first step is to find a way to prevent her from being able to reuse these lists, and the solution to that is called "salting". These hash lists already have to be generated separately for each hashing algorithm, but implementing different hashing algorithms for each user would be cumbersome for the developers of the server and its official client, and would introduce a lot of headaches. Instead, what if two users could enter the same password, and it would create a different hash even while using the same algorithm? That might *seem* like it would defeat the purpose of hashing - after all, since the hash generated from an entered password needs to be the same during sign up and every subsequent log-in. If the output was semi-random, the hash from the log-in couldn't be reliably matched with the hash in the database. However, rather than changing how the hash works, you can change how the input works!

Let's say that Jan uses their password *dragon*, and their friend Clyde who also works with this legal firm uses the same password, *dragon*. However, rather than just hashing *dragon* for both of them, upon signing up each user is assigned a unique random string (sequence of characters) called a "salt" that gets "sprinkled" in with their password to make their hash unique from others using the same password, and that salt gets stored in the database alongside the password hash. So, now when Jan signs up with *dragon*, they get assigned the salt *hotdog*, while Clyde gets assigned *burrito*. Now, Jan's hash takes *dragonhotdog* and outputs *f6a2c8632a1706f8c113e30db7cfc4ae*, while Clyde's takes *dragonburrito* and outputs *1d501a1b9a08124a75d29dd83ed68afd*. This way, both Jan and Clyde know the same secret, while the server knows the salt and the hash of the secret combined with the salt. So, Lilith might be able to see the salt and the hash if she compromises the database, but still couldn't know that both users had the same secret right away from just that information.

As a result, precalculated hash tables won't work on these passwords anymore, since those tables couldn't reasonably account for appending strings like *hotdog* and *burrito* to all of these common passwords. As such, Lilith needs to recalculate her hash tables for

[11] *The original RockYou breach mentioned here actually comes preinstalled on some Linux distributions.*

each account's assigned salt to make any headway. If she's able to leak all the user info from the database to get the hashes, she will also get the salt since they'll be stored in the same place (otherwise there would be no way for the server to combine the same salt with Jan's password each time), so she *can* generate a new list and guess Jan's password - but now it will take more time, because she has to generate a new list for *every* account she wants to crack.

To make that even more frustrating, we can use a hashing algorithm that implements a feature called "CPU Loading". Essentially, it takes time to perform a hashing operation, but faster CPUs can do them at those aforementioned rates of hundreds of millions per second, which makes guessing feasible. If we want guessing to be less feasible, we can make the hashing take longer - the easiest way being to make a singular output hash actually be the product of many, *many* rounds of hashing, which will eat up more time per outputted hash. It's a bit like how even if Lilith got Jan's hash, inputting it into the log-in would be useless, since it would hash the hash. CPU loading takes this to the extreme by doing *many* iterations - for example, a great hashing algorithm in modern use is bcrypt[12], which at an iteration cost of 12 requires a high enough number of iterations that it takes my Intel(R) Core(TM) i7-7700HQ half a second to perform a hash. There are certainly better processors out there, but even assuming that Lilith's is 1000 times better, she would only be able to perform 2000 hashes per second for that particular algorithm. Suddenly, testing millions (or billions) of possible passwords per person becomes *far* less appealing.

So, in addition to securely communicating with the server, Jan can feel safe in knowing that their password will at least be *extraordinarily* annoying for someone like Lilith to guess if it gets leaked. So, let's take a closer look at how Jan's authentication token actually gets handled: cookies.

## Cookies

You've probably heard of them - the little pop-ups when you visit a site asking if you consent to them. If you're from the EU, there's even a chance you've actually been able to say "no" on one of those prompts! However, while many cookies are used for targeting ads and otherwise tracking users, their other big use is for authentication - so what are they, actually?

Nothing very impressive, really. They're little text files that get returned by a server's HTTP response to certain requests, contained in a HTTP header called *Set-Cookie*. The browser then stores these text files, and will attach their information to any subsequent request made to the same server that returned the cookie. The *Set-Cookie* header contains a *name* and a *value*, which is all the cookie really is - a text value paired with a text label that gets saved by the browser for future use. Thus, an actual cookie might look like *token=JanIsValid83*, where *token* is the name, and *JanIsValid83* is the value. In

*[12]As a side note, bcrypt even comes with a built-in salt generating function, and requires a salt input which gets stored in the output hash - it's a nifty algorithm to use.*

coding terms, the text value and label are called "strings", which is a fancy word for a sequence of characters.

In addition to the name and value string pair, when an HTTP response header returns a cookie, there are several additional string pairs which it can add to the cookie which the browser will use to interpret how to store and handle the cookie. As such, these are predefined key-value pairs which have certain specifications that browsers are supposed to follow. Still no magic, just strings and a browser that has code intended to interpret certain strings in certain ways.

These predefined key-value pairs, called "attributes", usually have default values so one only *needs* to specify the *name* and *value* of a cookie. However, one *can* specify a *Domain* and/or *Path* to change which web addresses requests must target for this cookie to get attached to them. Additionally, the attributes *Expiration* or *MaxAge* can be set to tell the browser to delete the cookie at a specified date and time in the case of *Expiration*, or after a certain amount of time has elapsed with *MaxAge*. If neither of these is specified, the cookie will be deleted when the browser deems the session to be over, which might not be when you close the tab. Browsers tend to retain information as much as possible, and would prefer to "restore" your session with the saved authentication cookie rather than force you to re-authenticate - their primary goal is usually ease of use, not security.

Additionally, there are some attributes that can be set for cookies which are strictly for security - I'll explain them roughly for now, and discuss their practical applications later. First up is *SameSite*, which can be set at one of three levels: *Strict* - which means the cookie can only be sent if the request is initiated from the cookie's original domain, *Lax* - which means the cookie will be sent on *certain* requests initiated from a *different* site than its origin, or *None* - which means the cookie will be sent on *any* requests initiated from a different site than its origin. Another attribute, *Secure*, means the cookie will only be sent over HTTPS connections, not HTTP. Finally, *HttpOnly* tells the browser to store the cookie in a way that keeps it from being accessed by JavaScript (the most common web scripting language). Essentially, this marks the cookie as being solely for the use of the browser and the server - the client shouldn't ever touch it directly.

So, for those of you wondering how the server for the legal database was returning the authentication token to Jan, it was with cookies. Jan sends the log-in request with their credentials, *dragon* gets hashed, and the server matches that hash against the hash stored for the user *Jan*. If they match, it generates *JanIsValid27*, and returns the token in a cookie through an HTTP response that has a *Set-Cookie* header, which might look something like the following:

*Set-Cookie: authToken=JanIsValid27; MaxAge=1200; SameSite=Strict; HttpOnly; Secure*[13]

This tells the browser to store the cookie with the value "*JanIsValid27*" (with a few additional settings explained in the footnote). Now, whenever Jan makes a request to the legal database's server, the browser displaying their client program will automatically attach that cookie (*authToken=JanIsValid27*) to the request. This allows the server to check if any cookies containing the name *authToken* were sent, and if they were, to read the associated authentication token. This can then be checked for validity to authenticate the request as coming from Jan.

There are other ways to accomplish this, though. Rather than responding with a cookie, the server could reply with the token in the body of the HTTP response. Since this isn't sent with a *Set-Cookie* header the browser doesn't manage the token, thus making Jan's client program responsible for the token's storage and sending it with requests. However, this method requires some extra work from the designer of the client app, and in some cases can be less secure. Since the client app is written in JavaScript or a related language, the way it can store information in the browser is a chunk of memory set aside by the browser called "LocalStorage", which by definition is accessible to scripting languages. Since a number of attacks consist of an attacker inserting malicious JavaScript into an application (commonly called "injection"), storing the authentication token in LocalStorage might allow Lilith to access Jan's token much faster during an attack than if it were stored in an *HttpOnly* cookie. JavaScript can't directly access a cookie that's been set to *HttpOnly*, which should slow her down, but as will be discussed later, it's not *impossible* to use JavaScript to view an *HttpOnly* cookie. Remember, security is never *secure*.

## *Auth Tokens & JWT*

One question that arises is "what should actually be stored in this cookie?" - after all, "<username>IsValid<2DigitNumber>" isn't a particularly hard authentication token for an attacker to guess if they learn the format. Lilith might be able to try only 100 different requests to the server to hijack Jan's session! Remember, this authentication token carries the same authority with the server as their initial password exchange - it represents that this user knew a correct secret. So, even if Lilith doesn't know Jan's secret, if she gets their authentication token, she can *pretend* to have known it. What, then, would a good token look like?

In a traditional *stateful* cookie approach, when Jan logs in, the server would check their credentials, and if they're authenticated, it would create a "session" in the database with relevant information like IP address, username, time the session started, and so-on. This session info would all be stored in connection with a randomly generated id, which is normally a UUID (Universally

[13]*This cookie is a key-value pair of* authToken *and* JanIsValid27*, set to expire after 2 hours, only be sent on requests initiated from the origin. site, to be inaccessible to JavaScript, and to only be sent using an HTTPS connection.*

Unique Identifier, a 128-bit number which is so large that it is unlikely for duplications to occur) as the "address". This might look something like *c4fb98e8-acaa-42e5-9fa9-2fcad89f2e97*, and this UUID is what would be stored in the cookie that gets returned to Jan's client program.

Thus, whenever Jan sends a request to the legal server, the browser includes the UUID cookie with the request, so the server can check the database to see if this is a valid session, and return any important information about the session stored at that UUID. This approach is solid because a UUID is hard to guess and information about the session is fully controlled by the server and database. Furthermore, because the validity of the session is determined by the presence of the session info that's stored in the database, the session can be terminated at any point by the server deleting the info from the database, even before the cookie with the UUID expires. This offers a lot of control over the session, which is good, though it's not without downsides. For anyone developing an app, managing logout procedures to invalidate the session when users simply close the tab or window displaying the client, rather than click logout, is quite a bit of work. Additionally, actually maintaining the session info in the database can introduce some complexity for developers as well.

That brings us to the newer *stateless* method, where when Jan logs in, the server checks their credentials and then if they're authenticated, it would create a JWT (JSON[14] Web Token) to return to Jan's client as a cookie. This JWT is formed by three sections separated by periods - the first is the header with meta-information about how the token is constructed, the second section is any information about the session, and the third is a hash of the header and session info together with a secret key known only to the server (this process is called HMAC - Hash-based Message Authentication Code). Each of these sections are represented in Base64 (binary is base 2, decimal is base 10, and so-on), and while they may *look* encrypted, they are fully readable by whoever has the token so long as they can interpret Base64..

However, since the third section is a hash that involves a secret known only to the server, these tokens can't be created/guessed easily. Additionally, even if Lilith managed to steal a JWT for Jan's session, she can't change any of the info about the session - when the server tries to perform the HMAC of the header and session info with the secret key, if any of the input changed the hash would be different and authentication would fail. This stateless method is beneficial because it reduces the amount of work done by the server (database lookups [like for the UUID in the stateful approach] take a lot more effort than generating a hash) and reduces code complexity. However, even though JWTs *should* include an expiration date in the session info, there is less control over these stateless sessions because they can't be revoked without blacklisting a specific token. That can be difficult since

[14] *JavaScript Object Notation - it's a data format that's become a standard for sending information in web apps that's easy for JavaScript to parse.*

their stateless nature means they shouldn't already be stored anywhere in the database.

A JWT can be quite long, and looks like so:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
.eyJpc3MiOiJPbmxpbmUgSldUIEJ1aWxkZXI
iLCJpYXQiOjE2Njc3MjI0ODYsImV4cCI6MTY
2NzcyMzcwOSwiYXVkIjoiTXkgUmVhZGVycyI
sInN1YiI6Ik51bGwiLCJHcmVldGluZyI6Ikh
vd2R5IGZvbGtzISBIb3BlIHlvdSBoYXZlIGE
gbG92ZWx5IGRheS4ifQ.VBsMw9NUH1DI1nRm
VWOFoEipSqktnz4Z9BvDDSuWnnc
```
[15]

If the server's secret is guessed, Lilith can generate her own valid JWTs for the legal database's server, so the strength of the secret is incredibly important - something like a 2048 bit key should be the *minimum*, and it should be changed on occasion to minimize the opportunity for attackers trying to brute-force guess it.

## *CSRF & XST*

Now, about those security settings for cookies - we already touched on *HttpOnly* being used to prevent JavaScript from reading the authentication token, but what about *SameSite* and *Secure*?

*Secure* relates back to HSTS - ideally, information shouldn't be sent over an unsecured connection, and while HSTS is our best attempt to force insecure connections to upgrade to HTTPS, setting cookies to *Secure* forces the cookie to only be sent if the connection is *already* HTTPS. So, not only does this help keep cookies safe from being read in transit, but it can *also* encourage users to use an HTTPS connection for the rest of their traffic. After all, if the authentication cookie can only be sent over a secure connection, Jan isn't going to be able to do much beyond the log-in portal if their connection isn't secure - the server will only offer information to the rest of the client app if Jan upgrades to an HTTPS connection so the authentication cookie can be sent.

The other attribute, *SameSite*, takes a bit more depth to understand. One thing you may have noticed about cookies is that they get sent with every request - what if you make a request you didn't intend to? This is where CSRF (Cross-Site Request Forgery, sometimes also written XSRF) comes in.

A server and client aren't actually connected pieces of software in any meaningful way (normally) - the server is a program that has a number of commands it will respond to and an internet connection with a particular address people can request those commands through. The client is just a program written to make a user-friendly way of submitting these commands to the server, so websites are accessible to everyone without having to learn the unique server commands for each one. As a result, anyone can write their *own* client if they *do* bother to learn those commands.

Now, let's say that Lilith sets up a website that's just an informational page, but in the background she's hidden a bit of code that will make a request to the legal server to delete some incriminating documents from a cybercrime lawsuit against her. That request, without an authorization token, will get

[15]*This was generated with the (not so) secret key* correcthorsebatterystaplepuppymonkeybaby

rejected by the server. However, she sends a link to her website to Jan, who clicks it because it seems like a useful informational site. Now, their browser displays the page *and* executes the hidden code, which sends along the authorization cookie they have since they're logged in to the official client. Now, there's both a valid request *and* a valid authorization token, so the server performs the requested action without Jan's knowledge. Thus, Lilith has successfully destroyed what she wanted to. Even worse, this method could be used to send a request to update Jan's password!

One important note with these attacks - they normally cannot *steal* information. Lilith typically won't be actively watching Jan's network traffic when using a CSRF attack, and browsers don't allow scripts to access responses that they make to a different site.[16]

This is where *SameSite* shines - it has three different levels. *None* means the cookie can be sent from *any* site, and does nothing to combat CSRF. *Strict* means the cookie can be sent only from the site that assigned it (the URL displayed in the browser must be the same). *Lax* means the cookie can be sent if the request is from the same site that assigned it *or* if a "safe" request (certain types of HTTP requests are considered safer than others) is made in a way that would cause the browser to navigate to said site that assigned the cookie (actually changing the URL and displayed webpage). By setting a cookie to *SameSite: Strict*, Lilith's CSRF attack won't work anymore, since her website isn't in the same domain as the site that originally issued the authentication cookie, so the cookie won't be sent.

However, sometimes a cookie *will* need to be sent from a different site, in which case *SameSite: Strict* isn't feasible. Say, if you're a social media platform that needs users to be able to log-in, close the app and surf the web, then be immediately logged back in when they click a link - otherwise, folks will get annoyed and the platform won't be all that social. In these cases, one can use an anti-CSRF token, which is a random string generated in association with a user's session and stored in the database. This is then supplied to the client to insert into the body of their HTTP requests (rather than a cookie).

Alternatively, it can be automatically inserted by the server into any official page that can make an HTTP request when that page gets returned to the client. This token is unique and *only* sent via HTTP bodies (not just automatically included with every request like cookies are) - only requests generated by the official client program should contain it. So long as Lilith doesn't know this anti-CSRF token (because the client & server apps are using HTTPS and following all these other good practices!), she can't insert it into her malicious forms. So, even though she can trick someone with a valid cookie into making a request, the request will lack the anti-CSRF token and be rejected.

[16] *Well, strictly speaking there are exceptions. This security policy can be managed using* CORS *(Cross-Origin Resource Sharing) policies to allow limited access. But generally, these should only be used with great caution and knowledge.*

That said, this implementation only works for stateful authentication patterns since it requires storing a value in the database - for stateless patterns, there's a technique called "double submit cookie", and it works just like it sounds on the label. Since CSRF attacks don't actually involve stealing the cookie, just taking advantage of the cookie being sent on requests automatically, Lilith doesn't actually *know* the token stored in Jan's authentication cookie. As such, if Jan's client inserts the authentication token into the body of every HTTP request it makes, the server can look for the token in *both places*, and if it's only sent in a cookie, the server can reject the request in the same manner as if it didn't receive the anti-CSRF token. However, note that this does involve Jan's client holding the token, which as discussed earlier, can be problematic and is why cookies should be set to *HttpOnly* if they contain secrets like an authentication token.

Another way to accomplish this stateless style is for the server to make a hash of the authentication token using the server's secret key and return *that* as an anti-CSRF token. If the authentication token sent with the cookie is valid, and then the HMAC of that token and the server's secret key matches the hash sent in the HTTP body, the request is validated and returned. State avoided, and anti-CSRF precautions implemented without exposing the authentication token.

One final note on cookie security - just because a cookie is set to be *HttpOnly* doesn't mean it's wholly inaccessible to JavaScript. One particular type of attack dubbed XST (Cross Site Tracing) makes use of an HTTP request method called *TRACE*, which is a diagnostic request that prompts the server to respond with some information about the TRACE request, including any cookies sent along with it. This can be fixed by disabling TRACE requests from being accepted by your server, but it shows that, once again, for every security measure there are plenty of ways to get around it.

## Putting it All Together

So, for summary's sake, what would a *good* application's authentication look like?

The server would have an SSL certificate signed by a trusted CA installed, have HSTS enabled, and have TRACE requests disabled.

The client would allow Jan to input their username and password, and would hash Jan's password with a salt (likely based on their username) to reduce consequences of eavesdropping before passing it along to the server in an HTTP request. The server would hash this received hash using bcrypt at a sufficiently high cost to take around half a second for average modern devices, then retrieve Jan's stored hash (which was hashed using bcrypt at the same cost) by their supplied username to make a comparison.

If they don't match, an error message is returned that doesn't reveal unnecessary information, simply "invalid credentials" as a reason. If they do match, a stateful implementation would create session data stored in the database, keyed to a newly generated UUID, and return this UUID in a *Secure*, *HttpOnly* cookie with the strictest level of *SameSite* that's workable for the application and a somewhat short expiration, maybe 24 hours. In a stateless implementation, this would be the generation of a JWT token that has a short expiration (possibly as low as an hour) and is "signed" by an HMAC operation combining the header, body, and a server secret that's at least a 2048 bit random number, and then returned in a cookie with the same settings described above.

In the stateful implementation, the server should also supply a randomly generated number/string in an HTTP response body as an anti-CSRF token that's been stored in the session info, while the with the stateless implementation it should return an HMAC of the JWT with the server secret in an HTTP response as the anti-CSRF token.

Either implementation can also return a cookie with somewhat more relaxed precautions that contains any non-sensitive data like user preferences and a display name. All sensitive info should be retrieved through authenticated requests and not stored outside the database, but less sensitive info can be handled in a cookie like this.

Now, the client has a relatively secured connection with the server over HTTPS (due to HSTS and an inability to send the authentication cookie over a plain HTTP connection), which should expire and require re-authentication at an appropriate time for the application's usage.

## *Additional Vulnerabilities*

There are a few important vulnerabilities I wanted to mention that didn't naturally fit in elsewhere.

Injection is the first, and one of the scariest for developers - at some point, code is ultimately a string written by humans and interpreted by a computer into actions. As such, imagine Lilith has obtained valid credentials, and wants to steal every other user's password hash. The official client app has a function that allows users to enter a username and retrieve the first name and last name of that user, just so they can look up who's associated with each username in the office. The legal server accepts that username input in plain text from the client log-in and uses it to construct a query to the database, which then displays the results on the webpage. What if Lilith writes some code instead of a username?

For example, a common database querying language is called SQL (Structured Query Language), and a vulnerable query in a server might look like the following: *$query = "SELECT firstName, lastName FROM Users WHERE username == '$userInput'";*[17]

SQL databases typically store data in different tables for different types of "objects" (a coding term for a data representation of something - an example might be "users", which would have a username and password to represent them).

[17] *This particular example is taken from an old PHP server I tinkered with.*

Each row in the table represents a new instance of that object, and each column represents one element that comprises that object. So, the above query is asking the database to return the *firstName* and *lastName* column from every row in the table *Users* where the value in the *UserId* column matches some string *$userInput*, which is a variable containing a string that has been specified by the user. This server is *assuming* Lilith will write a username, like "Jan", which would let her see Jan's recorded first and last name. However, she could instead write something like "*1" AND 1 == 2 UNION SELECT username, passhash FROM Users;--*[18]

That probably looks like a bunch of nonsense to you - what the above is doing is filling the original query's need, getting rid of that original query's output, then replacing it with a query that retrieves the username and password hash for all rows in the table *Users*. It also negates the *rest* of the original query (which may or may not exist - Lilith doesn't see the query) so that Lilith's query is the only bit that gets executed. Now, by entering all this into the "username" field Lilith's client will show her a list of usernames where the first name should display and password hashes where the last name should display.

[18]*For those who are technically interested: in this query, the* "1" *is a dummy username for the query to match again, the* AND 1 == 2 *is a tautologically false statement that makes the query never actually return a row. The UNION statement is essentially performing a second query attached to the original query which Lilith nullified. This new query is a simple* SELECT *as described in the prior paragraph, and the -- is SQL's way of denoting a "comment", which is text that should not be interpreted as code.*

This may sound like a silly problem, but it takes forethought to avoid this kind of vulnerability, and with many programmers being self-taught, they may not have ever been exposed to this issue before and not address it. It's a relatively simple fix - there are some pre-made functions in most coding languages that handle user input to "sanitize" it, which essentially means either stripping out special characters or character sequences (like "=" or the quotes themselves), or making them harmless by changing them into another format that works *visually* for the user but doesn't function for the query language.[19]

A similar type of attack is XSS (cross-site scripting), which is the same underlying principle, but rather than "injecting" code into the server, code is injected into the client (whether that's directly through access to the client, or modification of what gets served to the client program by the server). This can be through user input that is then displayed on the web page, or even through 3rd party content like advertisements - sanitization is a *must* to avoid this.

Both injection and XSS attacks *completely* compromise an app's integrity. Not immediately, but given enough time, they can be leveraged into full control over both client and server, including access to all stored data. Yet, such an attack can be destructive and leave visible footprints, so

[19]*There are several ways to accomplish this, but a common one is to use "html entities", which are a set of codes to denote special characters in HTML (Hyper-Text Markup Language, the thing all webpages are made of). For example, = would be* &equals; *for the HTML entity. This will display as = in the browser, but SQL won't treat it as code.*

instead attacks often shoot for smaller targets like stealing authentication tokens. This is why we keep XSS in mind when designing authentication procedures - even though XSS of any level is a critical failure, and our first line of defense is input sanitization, we accept it as an inevitability when we design software because vulnerabilities *will* happen. So, we set our cookies to *HttpOnly* because we worry about XSS attacks, and want it to take them *that* much longer to find a way to hijack a session instead of just accessing the authentication token in a cookie through their injected JavaScript.

A final attack concept I want to address is "oracle" attacks - so named because they rely on gaining hidden information, much like the traditional idea of an oracle. These aren't so much a *formal* attack as a way to exploit vulnerabilities that, once again, simply aren't often considered in software design. You may have noticed in the prior description of a well-designed application that the server performs the hash of the submitted password *before* retrieving the hash stored in the database. That might seem strange, since it would be faster to retrieve the hash by the username first, just in case the submitted username was invalid and the server could reject the authentication at that point without having to do the work of hashing. However, what if Lilith wanted to get a list of usernames so she could see if those usernames show up in the leak from another database (people reuse usernames *and* passwords) so she could make an educated guess as to their password? If the server included that invalid username shortcut and didn't perform the half-second bcrypt hash, Lilith could enter username guesses and time the responses - invalid guesses would take half a second less time to resolve than valid ones. In this way, she could accurately construct a list of usernames (it would take a while, but is *possible* in a reasonable timeframe). This is also why the server should only return "invalid credentials" instead of "invalid username" or "invalid password", as this would also allow Lilith to guess valid usernames.

One formal attack that uses an oracle methodology is BREACH - an attack that can be used to "decrypt" messages sent over HTTPS. You may recall that I said HTTPS messages couldn't be *read* - however, they can be *guessed*. Essentially, most traffic uses something called "compression", wherein any part of the message which gets repeated is expressed in a shorter format. A simple example might be that the message "AAAAA" would be compressed into "Ax5" - it takes 2 fewer characters and saves on message size, only requiring some additional processing before and after it's sent. Since sending data over a network is usually *far* slower than any processing done locally on a computer, this is a massive time saver.

So, HTTP responses get compressed before they get encrypted to be sent over an HTTPS connection. What information can be gained with that? Well, Lilith could design another malicious website for a CSRF attack, and use it to send a massive number of guesses to the legal database server. Each of these guesses consists of a string, starting with a size of only 1 character. Now, Lilith uses another tool to monitor the size of the encrypted response - if it got longer by 1 character, the guess didn't match any of the plaintext in the response. However, if it stayed the same length, the guess was compressed, meaning that the character matched some of the plaintext in the response. Bam! Now Lilith's website can proceed to the next character, and do this over and over.

This *does* require a lack of CSRF protection, and some means to monitor Jan's network traffic despite the browser blocking Lilith's script's access to the responses. It's also limited to stealing information returned in the response body, which should rarely be secrets, though sometimes they are (say, the authentication token or anti-CSRF token). If these conditions are present, though, this attack is a real problem that gets around HTTPS connection's TLS encryption. Just because HTTPS solves a lot of problems, it's not a silver bullet and still relies on the support of other security measures.

## *Wrap-Up*

Once again, this paper is not comprehensive. I don't know every vulnerability, nor do I have the time and space to elaborate every single one I know here. Even if I could, by the time you read this, time has passed and new vulnerabilities have been discovered and new techniques invented. This field is constantly evolving, and it's nigh impossible for anyone to keep up.

You might be wondering why you should bother with any of this, then. Don't despair! Just because security practices can't make things completely impervious to attacks, and new tools for both offense and defense are being developed all the time, doesn't mean things are hopeless. Each of these practices outlined are *layers* of defense - the more you have the better, but they each serve a specific purpose with a tradeoff of time and energy for the particular protection they provide. If your site is static, pretty much *none* of these are necessary - if there's a log-in process, most of the password secrecy practices should be used to protect user data, but if the site doesn't handle things like commerce or medical information, you can probably get away without the rest.

Is it *better* if you use good practices with your cookies? Of course! But if you don't have much of anything that a hacker might want, your security is going to be tested far less because fewer attackers will care. Some of the easiest sites to crack out there, with plaintext passwords and no HTTPS, will probably stay untouched just because they're small and no one knows they exist who would want to attack them for the handful of passwords they store.

It's the age-old story of warfare. The safest suit of armor is one that encases you in solid metal - that makes it impractical, however, so you compromise and put joints in it, and that's full plate. But, full plate is expensive, so you also have chain mail which is a bit cheaper, but covers less and is prone to popping open if it gets stabbed too hard. There's even just thick fabric, which is far cheaper and accessible, and does a reasonable job of making cuts a lot less deep, even if it doesn't always stop them. Trade-offs. The physical security of these combatants is a lot like the digital security of web applications. With enough time and effort, you can get amazing protection with full plate, but you don't always have that time and effort to spend, so you can reasonably settle for chain or leather and still make it through most scrapes.

On top of that, *most* of these rapid changes are going to be pretty focused on the toughest security, so if that's not what your site needs, that's alright. You only need to worry about massive new attacks which just about invalidate one of the things you're using, like when attackers started figuring out that they could make hash tables ahead of time - that was a major change. On the flipside, BREACH is massively terrifying, but it requires such specific circumstances and is so hard to pull off that there's minimal information on it, and most people don't seem to care.

Going back to the armor analogy, full plate has a fascinating arms race against polearms that got better and better at puncturing it or just battering the person inside hard enough that they broke things inside their own armor. Most of these didn't matter much to the men-at-arms, though, just the knights who could buy the expensive full plate. Gunpowder, however, changed the game. It took a while, but once guns became reliable enough, not only did full plate drop out of favor, so did chainmail and most other types of armor. You couldn't do much to stop a bullet in those days, so they just decided to go light and hope it didn't hit them. Eventually, helmet designs were developed that could help a bit, and then kevlar made it so a fatal wound would just be a bad bruise, and even ceramic plates sometimes get used over vital areas if folks know they'll be up against some major firepower.

Ultimately, people still *survived* attacks countless times with suboptimal equipment, and people still *died* from attacks with the best. They used the best options they had the means for to up their odds of survival just a bit, and modern web security is the same. You should care because passwords, text messages, money, medical records, and so much more are all handled online now and *really* matter, even if they aren't all life and death. And you shouldn't throw your hands in the air just because you can't guarantee they'll be safe - the internet has, for better or for worse, made a lot of things in life far easier and really changed how things are.

There is risk to it, yes. There is no way to fully eliminate that risk, absolutely. But that doesn't mean we should throw it all away. We should take the best steps we can that make sense for us when building apps, or choosing which apps to use, to make things as low risk as possible, but not become paralyzed by the fact that something *may* go wrong.

That in mind, this paper provides a foundation to get you looking at the basics, and avoid some of the obvious flubs. Now you should have the tools to start to understand this war that's playing out all around you every time you log-in to a site or navigate to a new web page. Use these tools however makes sense for you, whether that's simply a new appreciation for how things work as a consumer of software, or as a springboard into the fray as a developer of software.

For further reading, I would highly recommend OWASP (Open Web Application Security Project) as a fairly authoritative source with lots of good information, and if nothing else, glance at their project OWASP Top 10, which identifies the top 10 most common security issues in web applications each year.

Go forth, know that security is flawed, but don't live in fear of that. Be informed, do more than the bare minimum, and hope that whatever you manage is enough that Lilith looks elsewhere for easier targets!