AppSec for Developers

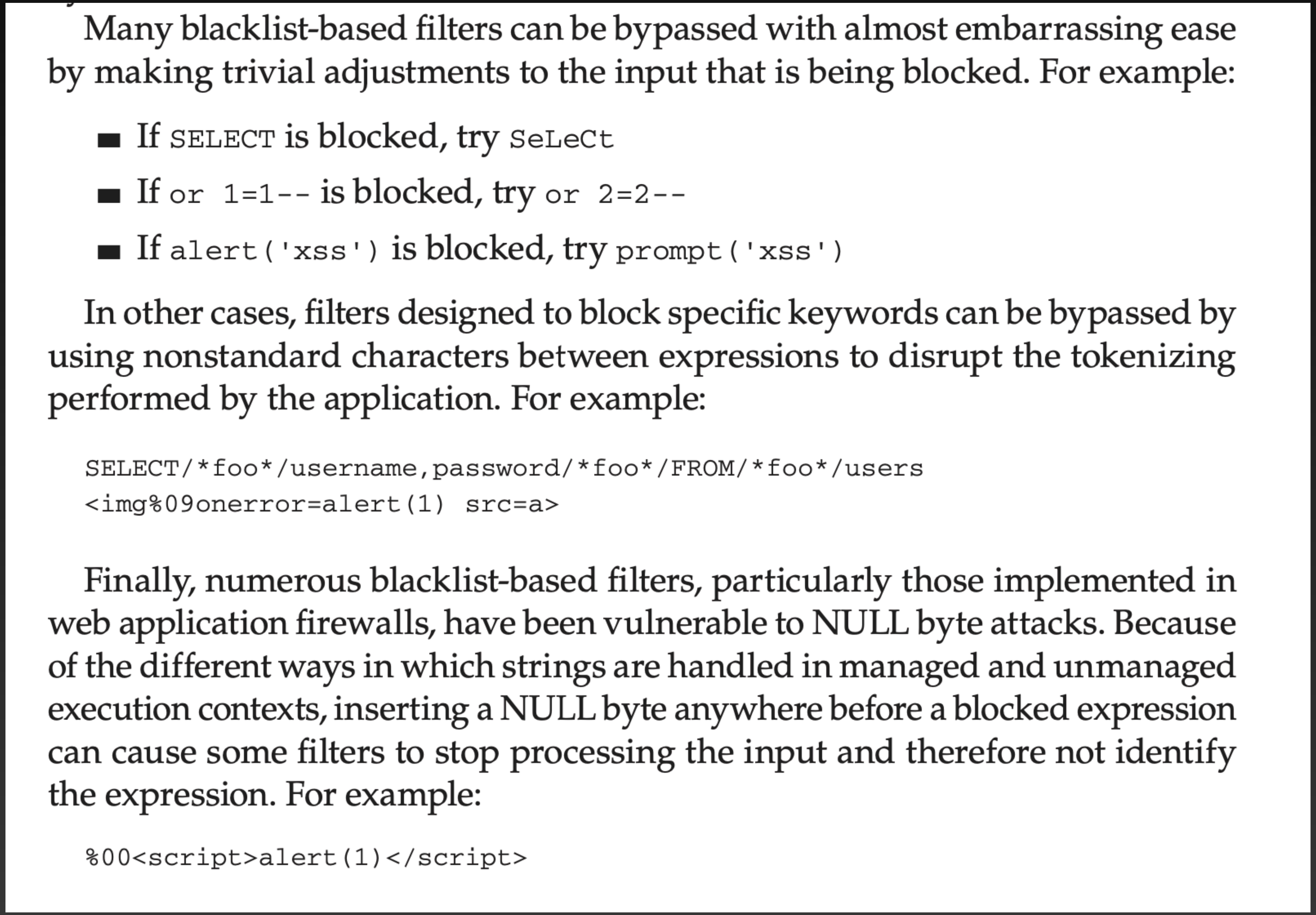
Introduction

Jawad NASSAR 2024-10-10

Introduction

assume all users are malicious

- Users Can Submit Arbitrary Numbers
  - HTML Form Price fields
  - Hidden fields
  - HTTP Requests are public and can be tampered with
  - Reject all known Bad inputs



Many blacklist-based filters can be bypassed with almost embarrassing ease by making trivial adjustments to the input that is being blocked. For example:

- If `SELECT` is blocked, try `SeLeCt`
- If `or 1=1--` is blocked, try `or 2=2--`
- If `alert('xss')` is blocked, try `prompt('xss')`

In other cases, filters designed to block specific keywords can be bypassed by using nonstandard characters between expressions to disrupt the tokenizing performed by the application. For example:

```
SELECT/*foo*/username,password/*foo*/FROM/*foo*/users
<img%09onerror=alert(1) src=a>
```

Finally, numerous blacklist-based filters, particularly those implemented in web application firewalls, have been vulnerable to NULL byte attacks. Because of the different ways in which strings are handled in managed and unmanaged execution contexts, inserting a NULL byte anywhere before a blocked expression can cause some filters to stop processing the input and therefore not identify the expression. For example:

```
%00<script>alert(1)</script>
```

Core Defence Mechanisms
sanitization

- **Sanitization**
    - This approach recognizes the need to sometimes accept data that cannot be guaranteed as safe. Instead of rejecting this input, the application sanitizes it in various ways to prevent it from having any adverse effects. Potentially malicious characters may be removed from the data, leaving only what is known to be safe, or they may be suitably encoded or "escaped" before further processing is performed.
    - Examples: **Safe Data Handling** (SQLi parameterized Queries) , **Semantic Checks** (confirm Account number belongs to user), **Boundary Validation** (confirm that input makes logical sense by the server-side)..

Core Defence Mechanism

sanitization

- Preventing XSS attacks may require the application to HTML-encode the > character as &gt;, and preventing command injection attacks may require the application to block input containing the & and ; characters.
- Attempting to prevent all categories of attack simultaneously at the application's external boundary may sometimes be impossible.
- A more effective model uses the concept of boundary validation. Here, each individual component or functional unit of the server-side application treats its inputs as coming from a potentially malicious source

Core Defence Mechanism
sanitization

- Common input handling issues
  - Application may block `<script>` tags but fails to skip `<scr<script>ipt>`
  - Similarly for if application removes ../ then ..\ then ....\/ would defeat the validation
  - Some technologies perform a “best fit” mapping of characters based on similarities in their printed glyphs. Here, the characters « and » may be converted into `<` and `>`, respectively, and `Ÿ` and `Â` are converted into `Y` and `A`.

Core Defense Mechanism

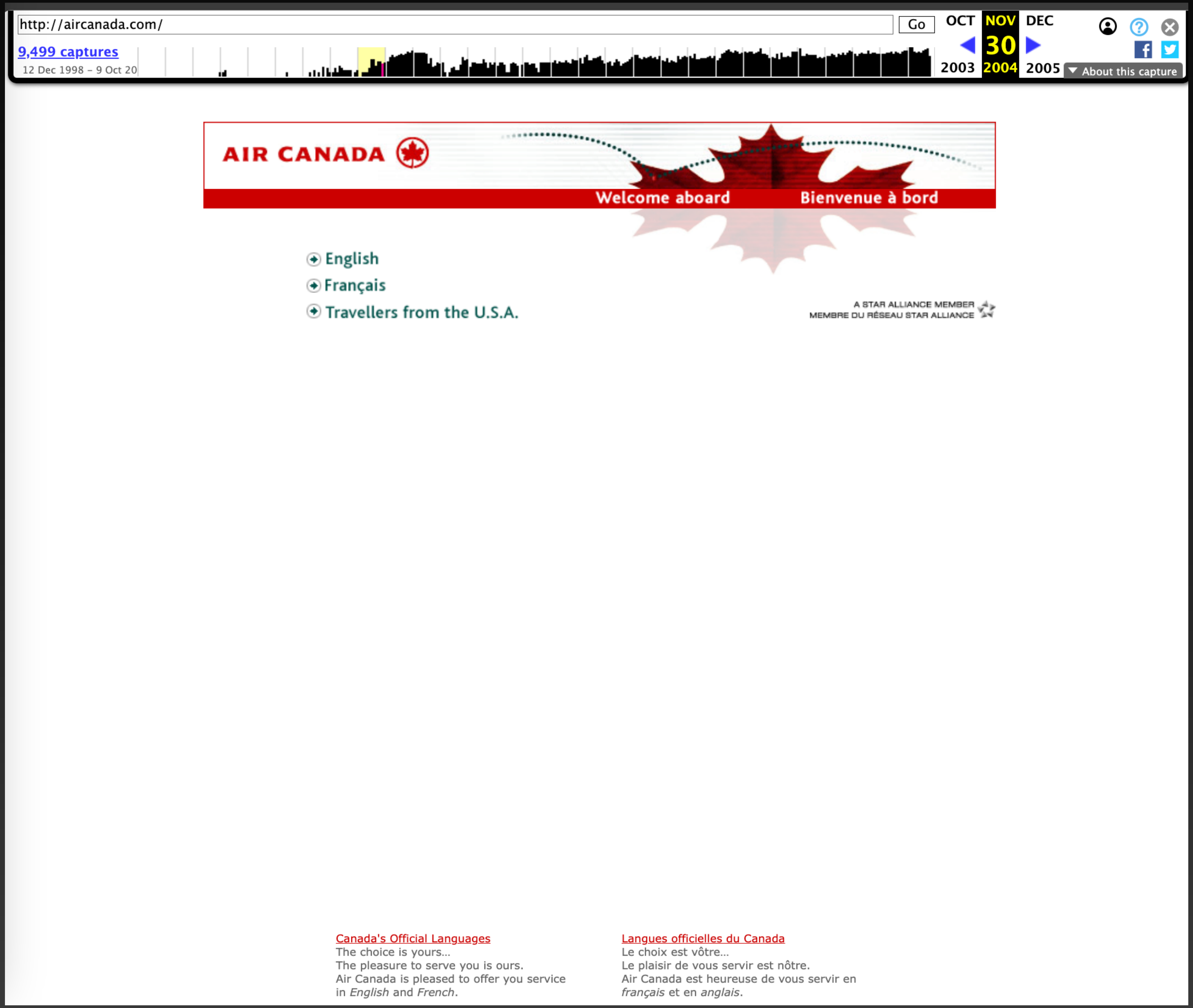
predictable patterns

- If there are pages called AddDocument.jsp and ViewDocument.jsp, there may also be pages called EditDocument.jsp and RemoveDocument.jsp.
- You can often get a feel for developers' naming habits just by reading a few examples. For example, depending on their personal style, developers may be verbose AddANewUser.asp, succinct AddUser.asp, use abbreviations AddUsr.asp, or even be more cryptic AddU.asp.
- Getting a feel for the naming styles in use may help you escalate your privilege if server side checks are not properly implemented.

Core Defense Mechanism

Public facing web apps, will be cached by search engines and wayback machine (internet archive), and can be used to attack it.

This is an example of how aircanada.com looked like in 2004-11-30



Canada's Official Languages
The choice is yours...
The pleasure to serve you is ours.
Air Canada is pleased to offer you service in *English* and *French*.

Langues officielles du Canada
Le choix est vôtre...
Le plaisir de vous servir est nôtre.
Air Canada est heureuse de vous servir en *français* et en *anglais*.

Bypassing Client-Side Controls

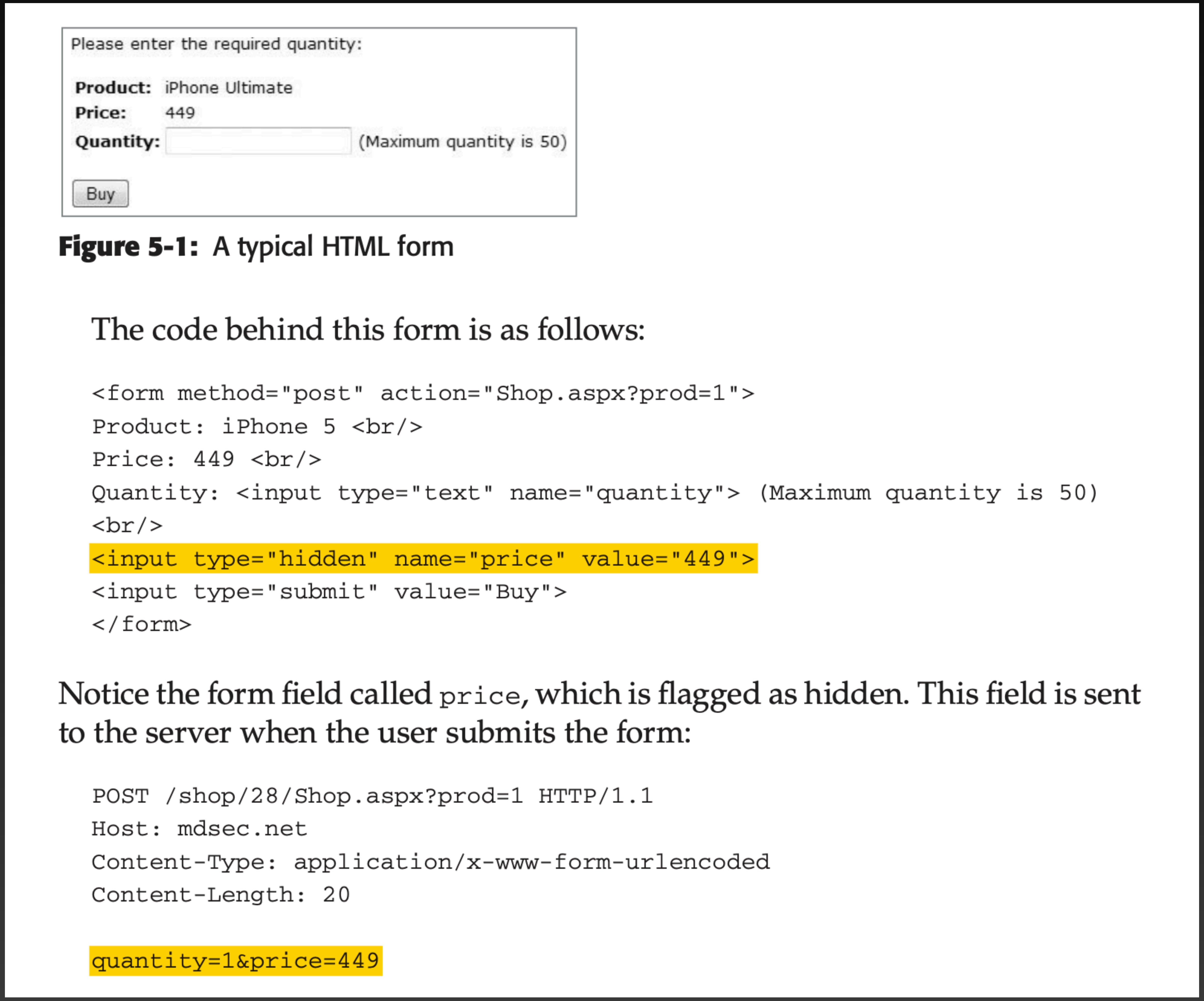
HTTP form & header risks

- If the server knows and specifies a particular item of data, the application wouldn't need to transmit this value to the client and then read it back. However, writing applications in this way is often easier for developers for various reasons:
  - **It removes the need to keep track of all kinds of data within the user's session**. Reducing the amount of per-session data being stored on the server can also improve the application's performance.
  - If the application is deployed on several distinct servers, with users potentially interacting with more than one server to perform a multistep action, it may not be straightforward to share server-side data between the hosts that may handle the same user's requests. Using the client to transmit data can be a tempting solution to the problem.
  - If the application employs any third-party components on the server, such as shopping carts, modifying these may be difficult or impossible, so transmitting data via the client may be the easiest way of integrating these.
  - In some situations, tracking a new piece of data on the server may entail updating a core server-side API, thereby triggering a full-blown formal change-management process and regression testing. Implementing a more piecemeal solution involving client-side data transmission may avoid this, allowing tight deadlines to be met.
- **However, transmitting sensitive data in this way is usually unsafe and has been the cause of countless vulnerabilities in applications!**

Bypassing Client-Side Controls

HTTP: form & header risks

- Hidden HTML Form Fields are not actually hidden
- HTTP Headers can be tampered with:
  - Set-Cookie HTTP Header (Cookies)
  - HTTP Post method parameters



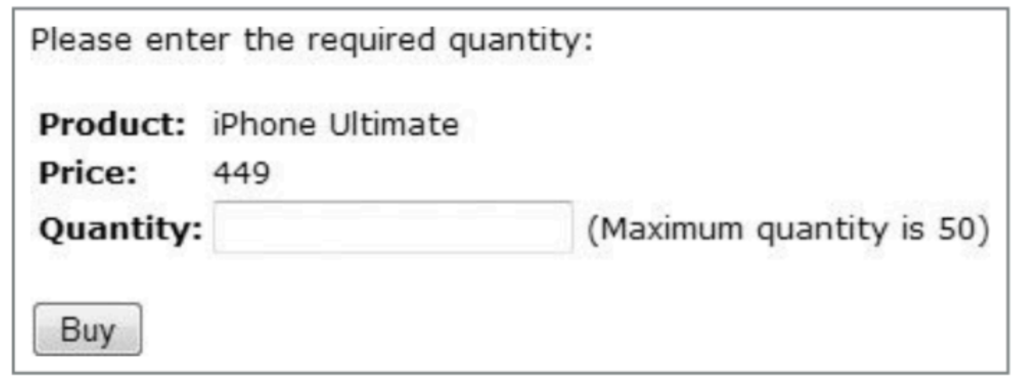


Figure 5-1: A typical HTML form

The code behind this form is as follows:

```
<form method="post" action="Shop.aspx?prod=1">
Product: iPhone 5 <br/>
Price: 449 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="hidden" name="price" value="449">
<input type="submit" value="Buy">
</form>
```

Notice the form field called `price`, which is flagged as hidden. This field is sent to the server when the user submits the form:

```
POST /shop/28/Shop.aspx?prod=1 HTTP/1.1
Host: mdsec.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 20

quantity=1&price=449
```

Bypassing Client-Side Controls

Opaque data

Sometimes, data transmitted via the client has been encrypted or obfuscated in some way. For example, instead of seeing a product's price stored in a hidden field, you may see a cryptic value being transmitted

The pricing_token parameter in the previously shown form may contain an encrypted version of the product's price. Although it is not possible to produce the encrypted equivalent for an arbitrary price of your choosing, you may be able to copy the encrypted price from a different, cheaper product and submit this in its place

```
<form method="post" action="Shop.aspx?prod=4">
Product: Nokia Infinity <br/>
Price: 699 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="hidden" name="price" value="699">
<input type="hidden" name="pricing_token" value="E76D213D291B8F216D694A34383150265C989229">
<input type="submit" value="Buy">
</form>
```

Bypassing Client-Side Controls

ASP.NET ViewState

- Opaque, hidden and enabled by default in asp.net apps
- Mostly used to enhance performance (preserves elements within the UI w/o the need to maintain that on the server side)
- Developers use it to Store arbitrary information across successive requests as follows:

```
string price = getPrice(productId);
ViewState.Add(“price”, price);
```

Bypassing Client-Side Controls

ASP.NET ViewState

The form returned to the user now looks something like this:

```
<form method="post" action="Shop.aspx?prod=3">
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwULLTE1ODcxNjkwNjIPFgIeBXByaWNlBQMzOTlkZA==" />
Product: HTC Avalanche <br/>
Price: 399 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="submit" value="Buy">
</form>
```

When the user submits the form, their browser sends the following:

```
POST /shop/76/Shop.aspx?prod=3 HTTP/1.1
Host: mdsec.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 77
__VIEWSTATE=%2FwEPDwULLTE1ODcxNjkwNjIPFgIeBXByaWNlBQMzOTlkZA%3D%3D&
quantity=1
```

Bypassing Client-Side Controls

ASP.NET ViewState

- Changing the ViewState parameter at random results in an error message
- The ViewState parameter is actually a Base64-encoded string that can be easily decoded & tampered with!
- It's advisable never to store any customized data within the ViewState.
- The option to enable the ViewState MAC should always be activated to stop ViewState tampering

Takeaway:

Encoding != Encryption

If something is gibberish, or obfuscated - it doesn't mean its secure!

Bypassing Client-Side Controls

HTTP Status Codes

All HTTP response status codes are separated into five classes or categories.

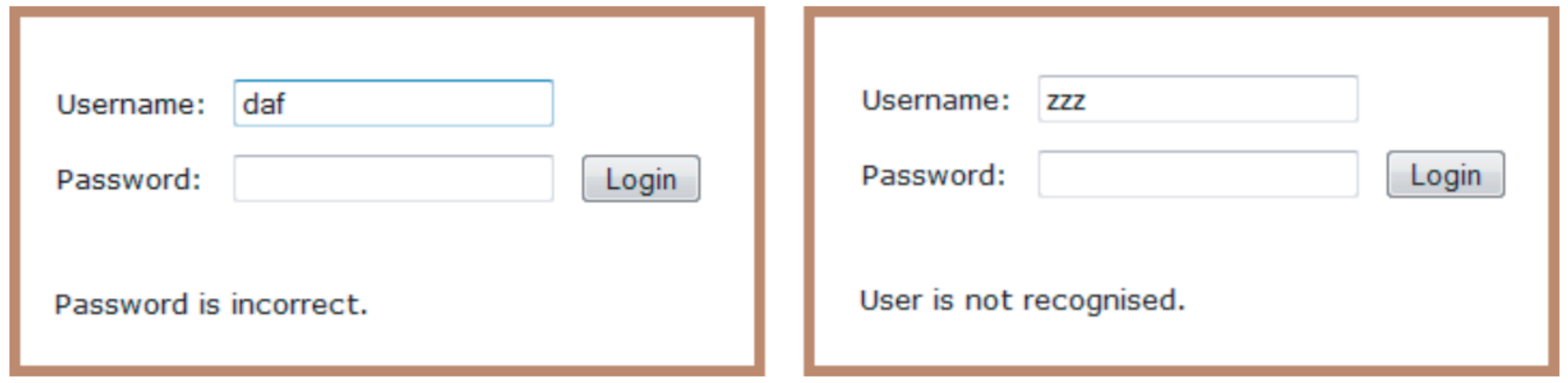
- 1xx informational response – the request was received, continuing process
- 2xx successful – the request was successfully received, understood, and accepted
- 3xx redirection – further action needs to be taken in order to complete the request
- 4xx client error – the request contains bad syntax or cannot be fulfilled
- 5xx server error – the server failed to fulfil an apparently valid request

Burp Suite Demo

Authentication

Common Vulnerabilities

- **Verbose Failure Messages:** Don't help attackers enumerate your application usernames and passwords



Username: daf
Password:
Login
Password is incorrect.

Username: zzz
Password:
Login
User is not recognised.

Figure 6-3: Verbose login failure messages indicating when a valid username has been guessed

Note: It may still be possible to enumerate usernames based on the time taken for the application to respond to the login request. Applications often perform very different back-end processing on a login request, depending on whether it contains a valid username.

Authentication
Common Vulnerabilities

Vulnerabilities that are deliberately avoided in the main login function often reappear in the password change function. Many web applications’ password change functions are accessible without authentication and do the following:

- Provide a verbose error message indicating whether the requested username is valid.
- Allow unrestricted guesses of the “existing password” field.
- Check whether the “new password” and “confirm new password” fields have the same value only after validating the existing password, thereby allowing an attack to succeed in discovering the existing password non-invasively.
- If the application generates an email containing a recovery URL in response to a forgotten password request, attackers obtain a number of these URLs, and attempt to identify any patterns that may enable them to predict the URLs issued to other users

Authentication

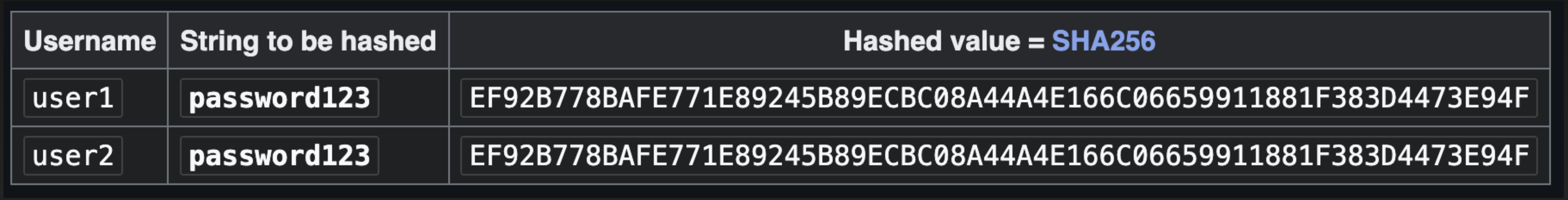
Insecure Storage of Credentials

- MD5, SHA1,.. and similar weak encryption or hashing algorithms is not recommended to use anymore, ensure to use up to date algorithms (preferably quantum resistant)
- Defer to SSO every time if its possible, and ensure end users cant Reset their passwords in that case scenario
- Salt is recommended to use to secure against Rainbow tables attacks
  - A rainbow table is a precomputed table for caching the outputs of a cryptographic hash function, usually for cracking password hashes.
  - In cryptography, a salt is random data fed as an additional input to a one-way function that hashes data, a password or passphrase. Salting helps defend against attacks that use precomputed tables (e.g. rainbow tables), by vastly growing the size of table needed for a successful attack.It also helps protect passwords that occur multiple times in a database, as a new salt is used for each password instance.

Authentication

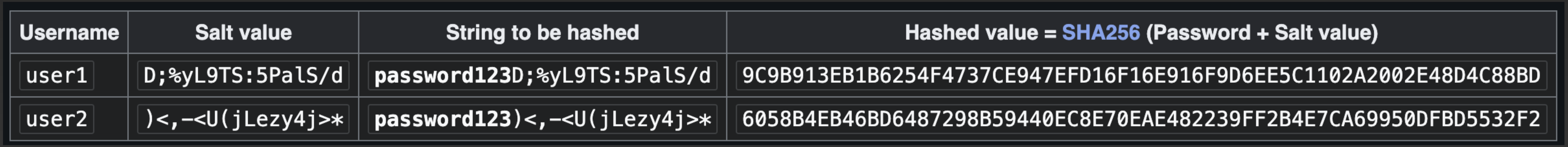
Insecure Storage of Credentials

- Without a salt, identical passwords will map to identical hash values, which could make it easier for a hacker to guess the passwords from their hash value.



Username	String to be hashed	Hashed value = SHA256
`user1`	**`password123`**	`EF92B778BAFE771E89245B89ECBC08A44A4E166C06659911881F383D4473E94F`
`user2`	**`password123`**	`EF92B778BAFE771E89245B89ECBC08A44A4E166C06659911881F383D4473E94F`

- Instead, a salt is generated and appended to each password, which causes the resultant hash to output different values for the same original password.



Username	Salt value	String to be hashed	Hashed value = SHA256 (Password + Salt value)
`user1`	`D;%yL9TS:5PalS/d`	**`password123`**`D;%yL9TS:5PalS/d`	`9C9B913EB1B6254F4737CE947EFD16F16E916F9D6EE5C1102A2002E48D4C88BD`
`user2`	`)<,-<U(jLezy4j>*`	**`password123`**`)<,-<U(jLezy4j>*`	`6058B4EB46BD6487298B59440EC8E70EAE482239FF2B4E7CA69950DFBD5532F2`

Authentication

Validate Credentials Properly

- Application should be aggressive in defending itself against unexpected events occurring during login processing.
- Application should use catch-all exception handlers around all API calls.
- Application should explicitly invalidate the current session, thereby causing a forced logout by the server even if authentication is somehow bypassed

Authentication
Session Management

- The server's first response to a new client contains an HTTP header like the following:

```
Set-Cookie: ASP.NET_SessionId=mza2ji454s04cwbgwb2ttj55
```

- Subsequent requests from the client contain this header:

```
Cookie: ASP.NET_SessionId=mza2ji454s04cwbgwb2ttj55
```

- The vulnerabilities that exist in session management mechanisms largely fall into two categories:
  - Weaknesses in the generation of session tokens
  - Weaknesses in the handling of session tokens throughout their life cycle

Authentication
Session Management

- For example, the following token may initially appear to be a long random string:

```
757365723d6461663b6170703d61646d696e3b646174653d30312f31322f3131
```

- However, on closer inspection, you can see that it contains only hexadecimal characters. Guessing that the string may actually be a hex encoding of a string of ASCII characters, you can run it through a decoder to reveal the following:

```
user=daf;app=admin;date=10/09/11
```

Authentication

Session Management

- **Weak Token Generation:** beware of adding timestamps to randomize tokens, as it is a very common and predictable practice

  ```
  String sessId = Integer.toString(s_SessionIndex++) +"-" + System.currentTimeMillis();
  ```

- **Weak Random Number Generation:** <u>Very little that occurs inside a computer is random</u>. Therefore, when randomness is required for some purpose, software uses various techniques to generate numbers in a pseudorandom manner. Some of the algorithms used produce sequences that appear to be stochastic and manifest an even spread across the range of possible values. Nevertheless, they can be extrapolated forwards or backwards with perfect accuracy by anyone who obtains a small sample of values.

Authentication
Session Management

Weak Random Number Generation: java.util.Random

```
synchronized protected int next(int bits) {
  seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
  return (int)(seed >>> (48 - bits));
}
```

This algorithm takes the last number generated, multiplies it by a constant, and adds another constant to obtain the next number. The number is truncated to 48 bits, and the algorithm shifts the result to return the specific number of bits requested by the caller.

Knowing this algorithm and a single number generated by it, we can easily derive the sequence of numbers that the algorithm will generate next. With a little number theory, we also can derive the sequence that it generated previously. This means that an attacker who obtains a single session token from the server can obtain the tokens of all current and future sessions.

Authentication

Session Management

- **Tests to verify your Application Session Management practices:**
  - Log in to the application twice using the same user account, either from different browser processes or from different computers. Determine whether both sessions remain active concurrently. If so, the application supports concurrent sessions, enabling an attacker who has compromised another user’s credentials to make use of these without risk of detection.
  - Log in and log out several times using the same user account, either from different browser processes or from different computers. Determine whether a new session token is issued each time or whether the same token is issued each time you log in. If the latter occurs, the application is not really employing proper sessions.
  - If tokens appear to contain any structure and meaning, attempt to sepa- rate out components that may identify the user from those that appear to be inscrutable. Try to modify any user-related components of the token so that they refer to other known users of the application, and verify whether the resulting token is accepted by the application and enables you tomasquerade as that user.

Authentication

Session Management

- **Vulnerable Session Termination**
  - In some cases, the logout function does not actually cause the server to invalidate the session. The server removes the token from the user's browser (for example, by issuing a Set-Cookie instruction to blank the token). However, if the user continues to submit the token, the server still accepts it.
  - In the worst cases, when a user clicks Logout, this fact is not communicated to the server, so the server performs no action. Rather, a client-side script is executed that blanks the user's cookie, meaning that subsequent requests return the user to the login page. An attacker who gains access to this cookie could use the session as if the user had never logged out.

Authentication

JWT

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.
- JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.
- In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:
  - Header
  - Payload
  - Signature
- Therefore, a JWT typically looks like the following.
- xxxxx.yyyyy.zzzzz
- Each part is a base64-encoded JSON object

Authentication

JWT - Header part

- The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
- For example:

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

- Then, this JSON is Base64Url encoded to form the first part of the JWT.

Authentication

JWT - Payload part

- The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.
  - **Registered claims:** These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: iss (issuer), exp (expiration time), sub (subject), aud (audience), and others.
    Notice that the claim names are only three characters long as JWT is meant to be compact.
  - **Public claims:** These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.
  - **Private claims:** These are the custom claims created to share information between parties that agree on using them and are neither registered or public claims.
- An example payload could be:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

- The payload is then Base64Url encoded to form the second part of the JSON Web Token.

Do note that for signed tokens this information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

Authentication
JWT - Signature part

- To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.
- For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

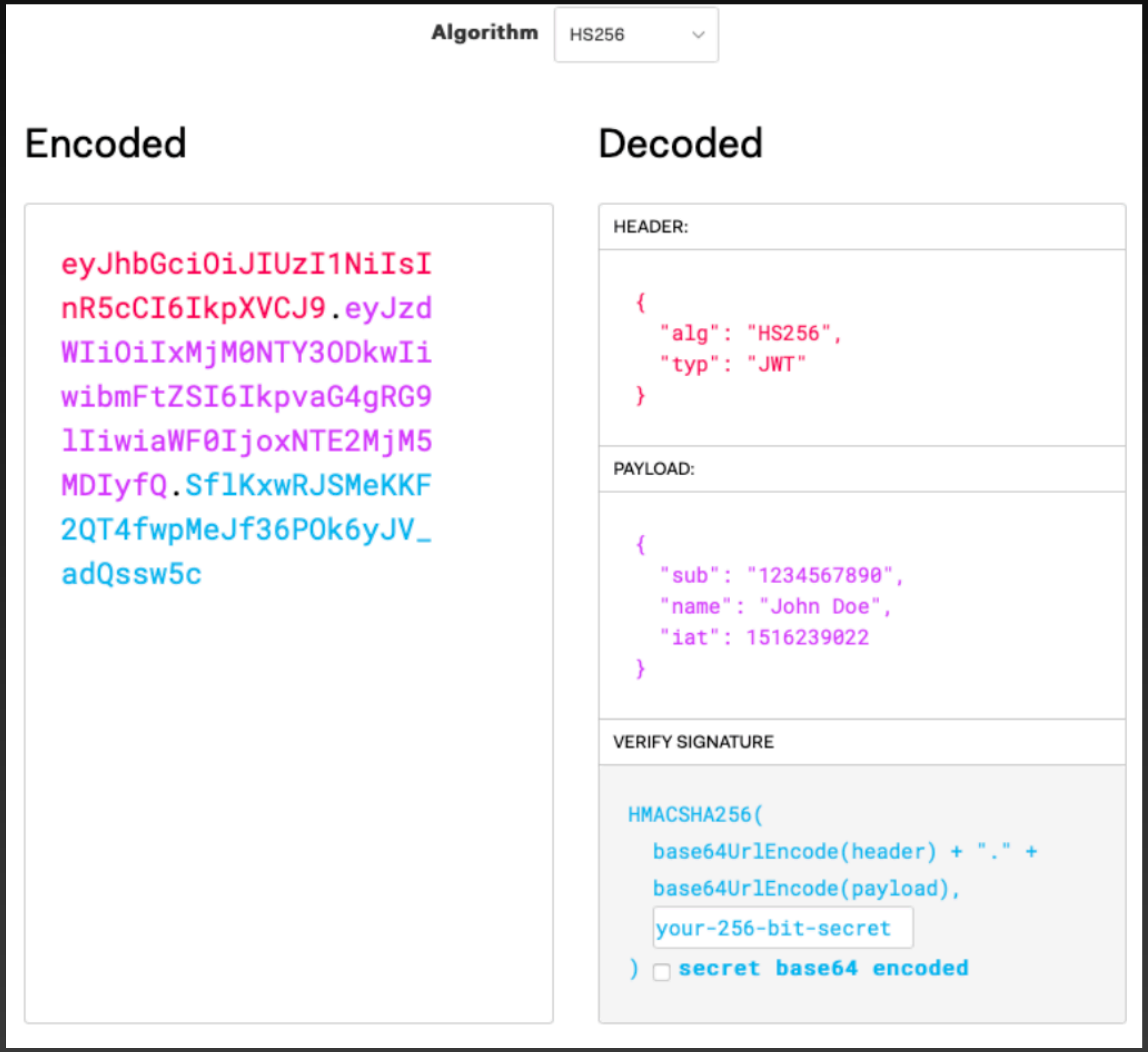
```
HMACSHA256(
   base64UrlEncode(header) + "." +
   base64UrlEncode(payload),
   secret)
```

- The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

Authentication

JWT - putting it all together

The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.


Algorithm HS256

Encoded

```
eyJhbGciOiJIUzI1NiIsI
nR5cCI6IkpXVCJ9.eyJzd
WIiOiIxMjM0NTY3ODkwIi
wibmFtZSI6IkpvaG4gRG9
lIiwiaWF0IjoxNTE2MjM5
MDIyfQ.SflKxwRJSMeKKF
2QT4fwpMeJf36POk6yJV_
adQssw5c
```

Decoded

HEADER:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Authentication

JWT - putting it all together

- Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema.

  ```
  Authorization: Bearer <token>
  ```

- This can be, in certain cases, a stateless authorization mechanism. The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. If the JWT contains the necessary data, the need to query the database for certain operations may be reduced, though this may not always be the case.
- Missing Signature, allows attackers to manipulate the data within the JWT payload

Note: Make sure you understand the difference between Stateless vs Stateful authentication

Access Controls

Insecure Access Control Methods

- **Multistage Functions:** The application should revalidate all fields at all stages, an attacker can intercept the final POST request of a specific page
- **Platform Misconfiguration:** Many backend endpoints allow GET and POST requests, however only validates and check on POST
- Parameter-Based Access Control:

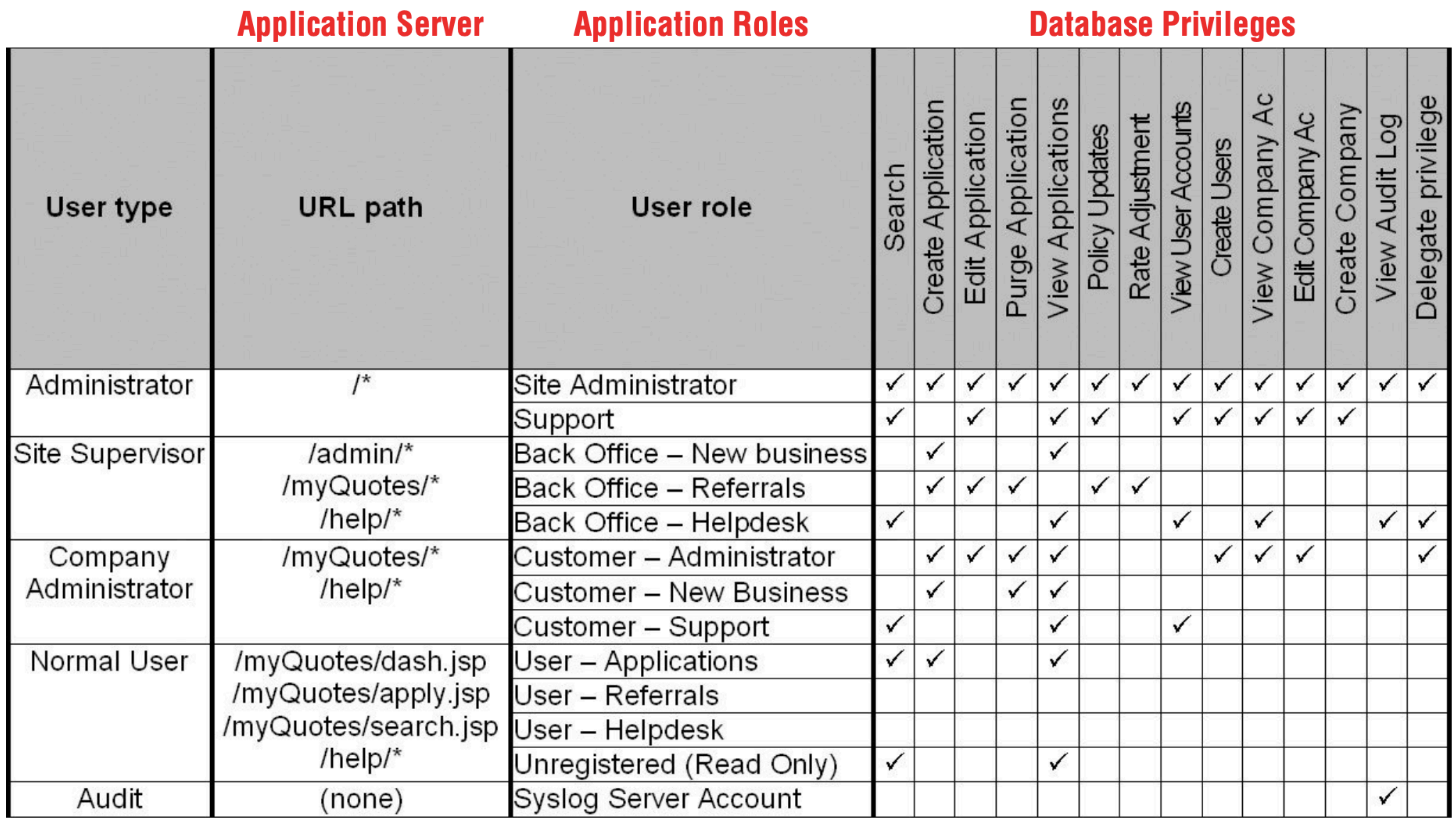
```
e.g example.com/login/home.asp?admin=true
```

- Don't assume that if a user reaches a specific page, that means that they arrived via legitimate means

Access Controls

Securing Access Control

- Do not rely on users ignorance of application URLs..
- Do not assume users will open pages in the intended sequence
- Drive all access control decisions from users sessions
- Log every Event where sensitive data Is accessed or sensitive action is performed
- Use different DB accounts and least privileges grants whenever its possible
- Always Build a privilege matrix for complex applications



	Application Server	Application Roles	Database Privileges													
User type	URL path	User role	Search	Create Application	Edit Application	Purge Application	View Applications	Policy Updates	Rate Adjustment	View User Accounts	Create Users	View Company Ac	Edit Company Ac	Create Company	View Audit Log	Delegate privilege
Administrator	/*	Site Administrator	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		Support	✓		✓		✓	✓		✓	✓	✓	✓	✓		
Site Supervisor	/admin/* /myQuotes/* /help/*	Back Office – New business		✓			✓									
		Back Office – Referrals		✓	✓	✓		✓	✓							
		Back Office – Helpdesk	✓				✓			✓		✓			✓	✓
Company Administrator	/myQuotes/* /help/*	Customer – Administrator		✓	✓	✓	✓				✓	✓	✓			✓
		Customer – New Business		✓		✓	✓									
		Customer – Support	✓				✓			✓						
Normal User	/myQuotes/dash.jsp /myQuotes/apply.jsp /myQuotes/search.jsp /help/*	User – Applications	✓	✓			✓									
		User – Referrals														
		User – Helpdesk														
		Unregistered (Read Only)	✓				✓									
Audit	(none)	Syslog Server Account													✓	

Figure 8-6: A privilege matrix for a complex application

References

- The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition Dafydd Stuttard, Marcus Pinto
- https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
- https://en.wikipedia.org/wiki/Rainbow_table
- https://en.wikipedia.org/wiki/Salt_(cryptography)
- https://en.wikipedia.org/wiki/JSON_Web_Token
- https://jwt.io/introduction/

Recommended Resources

- The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition Dafydd Stuttard, Marcus Pinto
- The Tangled Web: A Guide to Securing Modern Web Applications Michal Zalewski
- https://owasp.org/www-project-developer-guide/
- https://cheatsheetseries.owasp.org/

Next Session

Please vote on next Topic

- Workshop: Deep Dive into OWASP Top 10 (XSS, SQLi, Broken Access control, IDOR, SSRF, Command Injection etc..) - w/ Live application
- Crypto (Encryption, Hashing, Password Cracking) - w/ Live Workshop
- Shift Left (SCA, SAST, DAST) w/ Live Pipelines workshop
- Windows Server Security & Privilege Escalation risks (IIS Hardening)
- Authentication Deep Dive - JWT Risks live workshop
- More Web Applications risks (Local/Remote File inclusion) and other risks
- WAF Bypasses
- API Security
- More Burp Suite and similar tools features - w/ Live workshop

Q&A