Unified Singular Protocol Flow for OAuth (USPFO) Ecosystem

Jaimandeep Singh[a], Naveen Kumar Chaudhary[a]

[a]National Forensic Sciences University, Gandhinagar, India

ARTICLE INFO

Keywords:
OAuth 2.0
client impersonation
Secure Architecture
Vulnerabilities
DPoP, PKCE, JWS, PAR

ABSTRACT

OAuth 2.0 is a popular authorization framework that allows third-party clients such as websites and mobile apps to request limited access to a user's account on another application. The specification classifies clients into different types based on their ability to keep client credentials confidential. It also describes different grant types for obtaining access to the protected resources, with the authorization code and implicit grants being the most commonly used. Each client type and associated grant type have their unique security and usability considerations. In this paper, we propose a new approach for OAuth ecosystem that combines different client and grant types into a unified singular protocol flow for OAuth (USPFO), which can be used by both confidential and public clients. This approach aims to reduce the vulnerabilities associated with implementing and configuring different client types and grant types. Additionally, it provides built-in protections against known OAuth 2.0 vulnerabilities such as client impersonation, token (or code) thefts and replay attacks through integrity, authenticity, and audience binding. The proposed USPFO is largely compatible with existing Internet Engineering Task Force (IETF) Proposed Standard Request for Comments (RFCs), OAuth 2.0 extensions and active internet drafts.

1. Introduction

OAuth 2.0 is an open standard authorization framework that enables third-party applications to obtain limited access to a web service without sharing the resource owner's credentials. It is widely adopted and supported by many companies, libraries, and frameworks, and is used to protect web APIs and grant limited access to third-party applications Walls (2022).

Some of the most common uses of OAuth include social media platforms, such as Facebook and Twitter, to allow resource owner's (user's) to grant third-party applications access to their personal data and resources without sharing their login credentials. Identity and Access Management (IAM) systems to allow limited access to resources for authenticated users. Healthcare applications to securely share patient data between different systems and providers. This allows for better coordination of care and improved patient outcomes. Enterprises provide secure access to company resources and data to employees and third-party applications through OAuth Richer and Sanso (2017).

The process by which resource owners delegate access to their resources to the third-party applications instead of sharing their login credentials is also known as "delegated authorization". The temporary tokens, also known as access tokens, are issued by the authorization server, and are used by the third-party application to access the protected resources of the resource owner (user). The access tokens are issued with a specific scope, which defines the level of access that the third-party application is granted Siriwardena (2020).

1.1. Motivation

The OAuth 2.0 specification, by design, is rather nebulous and adaptable which allows for the emergence of poor practices leading to security vulnerabilities. The lack of built-in security features in OAuth 2.0 also means that the security relies heavily on developers using the right combination of configuration options and implementing their own additional security measures, which can be difficult for those inexperienced with OAuth 2.0 Portswigger (2023). The OAuth 2.0 specification also differentiates between public and confidential clients, resulting in different grants for each type, which leads to increased complexity and variability. This also causes issues with security and scalability as separate code bases must be maintained for different client types.

1.2. Our Research

In our research, we analyzed various vulnerabilities impacting the OAuth 2.0. We also studied the latest Proposed Standard Request for Comments (RFCs), OAuth extensions and active Internet Drafts proposed, updated and maintained by oauth and other working groups of Internet Engineering Task Force (IETF) for improving the security and architecture of OAuth.

Thereafter, we proposed a *Unified Singular Protocol Flow for OAuth (USPFO)* that is common for both confidential and public client applications. It would also ensure consistency and security across all implementations and help to minimize the potential vulnerabilities by reducing the variability and need for maintenance of different codebases in the OAuth implementation.

A *USPFO* is a standardized approach to implementing the OAuth protocol that takes into account various OAuth extensions and best practices to provide out-of-the-box security measures against common vulnerabilities such as client impersonation, token (or code) thefts, token manipulation and replay attacks. It also supports and incorporates use of OAuth extensions such as JWT-Secured Authorization Request (JAR) RFC 9101 Sakimura, Bradley

jaimandeep.phdcs21@nfsu.ac.in (J. Singh)
ORCID(s): 0000-0001-6266-1275 (J. Singh)

and Jones (2021), OAuth 2.0 Pushed Authorization Requests (PAR) RFC 9126 Lodderstedt, Campbell, Sakimura, Tonge and Skokan (2021), Proof Key for Code Exchange by OAuth Public Clients (PKCE) RFC 7636 Sakimura, Bradley and Agarwal (2015), Active Internet Drafts such as Demonstrating Proof-of-Possession at the Application Layer (DPoP) Fett, Campbell, Bradley, Lodderstedt, Jones and Waite (2023), and other best practices to improve the security of the OAuth flow. The tokens are cryptographically signed and bound to a specific audience, which helps to prevent token theft and replay attacks. The cryptographic signatures also ensure authenticity and integrity for the exchanged artifacts in the protocol flow. The *USPFO* is not a guarantee to solve all the vulnerabilities, but it is a step forward to minimize them.

The paper is organized as follows: In Section 2, the background of OAuth and it's commonly known vulnerabilities are discussed. The key features of USPFO and its comparison with traditional OAuth 2.0 approach are outlined in Section 3, followed by a detailed description of the protocol flow in Section 4. The various actors and their interactions are covered in Section 5. Section 6 addresses the security concerns associated with the unified flow. The paper concludes with a summary and future work in Section 7 and 8 respectively.

2. Literature Survey

2.1. Background

The OAuth protocol was developed by a small community of web developers from various websites and internet services to address the common problem of enabling delegated access to protected resources. In October 2007, OAuth 1.0, the initial stable release of the OAuth protocol, was made available. OAuth 1.0 revision A, which was released in June 2009, sought to strengthen the protocol's security and remove some ambiguities from the first iteration Hammer-Lahav (2010).

2.2. OAuth 2.0

OAuth 2.0, the current version of the OAuth protocol, was published as an RFC 6749 in 2012 Hardt (2012). OAuth 2.0 introduced several significant changes from OAuth 1.0, including a simpler, more flexible architecture and a focus on providing access tokens rather than sharing login credentials. It built upon the OAuth 1.0 deployment experience but it is not backwards compatible with OAuth 1.0.

OAuth 2.0 also introduced the concept of "bearer tokens", [Bearer Token Usage as RFC 6750 Jones and Hardt (2012)] which allow third-party applications to access resources without having to authenticate the user.

2.2.1. Grant Types

OAuth supports four different grant types, each suited to a different client application type. The two most commonly used grant types are:

Authorization Code Grant: This grant type is used by *confidential* clients such as server-side web applications. The confidential clients are capable of maintaining the confidentiality of their client credentials (such as a client secret). There are three steps in the authorisation code grant type. The user is first redirected by the authorization server to grant access to the third-party (client) application through User Agent (typically a web browser). The user is then redirected back to the client application with an authorization code. The client application can exchange the authorization code for an access token by sending it to the authorization server along with its client credentials Richer and Sanso (2017).

Implicit Grant: This grant type is similar to the authorization code grant type, but it does not require the third-party application to exchange the authorization code for an access token. Instead, the access token is returned directly to the third-party application, which can then use it to access the protected resources of the resource owner. This grant type is optimized for *public* clients, which do not have the ability to keep a secret, such as in a JavaScript-based application running in a browser, or in a native mobile app. Public clients are considered less secure than confidential clients that can prove their identity with a client secret Carnell and Sánchez (2021).

2.3. Prevalent vulnerabilities

Some of the prevalent vulnerabilities that can occur during the implementation of OAuth are listed below. These vulnerabilities are well-known and have been identified in many OAuth implementations in the past HackerOne (2023), Bugcrowd (2023), Portswigger (2023), Philippaerts, Preuveneers and Joosen (2022), Fett, Küsters and Schmitz (2016), Singh and Chaudhary (2022).

Improper validation of redirect URIs: An attacker can exploit an insecure redirect URI by intercepting the authorization code and using it to gain unauthorized access to the user's resources. Maliciously crafted URI and various URI manipulation techniques are used by attackers to steal the authorization code. This vulnerability can be mitigated by using a secure redirect URI (i.e. using HTTPS) and validating the redirect URI to match the one registered with the authorization server CVE-2021-43777 (2021), CVE-2022-36087 (2022), CWE-601: URL Redirection to Untrusted Site ('Open Redirect') (2022), CWE-20: Improper Input Validation (2022).

Improper use of the state parameter: If the state parameter is nonexistent or a static value that never changes, the authorization server cannot determine if the request was initiated by the client that generated the request or by an attacker. This allows the attacker to launch a CSRF (Cross-Site Request Forgery) by sending them a malicious link that initiates an authorization request with the attacker's own client ID to gain unauthorized access to the user's resources CVE-2022-43693 (2022), CVE-2023-24428 (2023), CWE-352: Cross-Site Request Forgery (CSRF) (2022), Arshad, Benolli and Crispo (2022).

Token leakage: It is a common vulnerability in OAuth implementations which can happen if an attacker intercepts the tokens (authorization code, access token or refresh token) that are being sent from the authorization server to the client or from the client to the protected resource server. This may occur if the connection between the entities is not secure, such as using an unencrypted connection (HTTP instead of HTTPS) or if the attacker is able to perform a man-in-the-middle (MITM) attack CVE-2022-32217 (2022), CVE-2022-32227 (2022), CVE-2022-39222 (2022), CVE-2023-22492 (2023), CWE-200: Exposure of Sensitive Information to an Unauthorized Actor (2022), CWE-312: Cleartext Storage of Sensitive Information (2022), CWE-319: Cleartext Transmission of Sensitive Information (2022), CWE-532: Insertion of Sensitive Information into Log File (2021), CWE-613: Insufficient Session Expiration (2022).

Client impersonation: The client secrets are used to authenticate the client to the authorization server and should be kept confidential. If an attacker is successful in obtaining the client secrets, they can impersonate the client and obtain access tokens by using the client secret and authorization code. Malicious client can impersonate as a genuine client to gain additional restricted scopes from the authorization server that would otherwise not be granted to a less trustworthy application. This can give the attacker access to more sensitive user data or resources. To mitigate this vulnerability, the client secrets should be stored in a secure location and protected with encryption. Additionally, access to the client secrets should be restricted to only those who need it. It's also important to rotate the client's secret frequently CVE-2022-31186 (2022), CVE-2019-5625 (2019), CWE-522: Insufficiently Protected Credentials (2022), CWE-532: Insertion of Sensitive Information into Log File (2021), CWE-922: Insecure Storage of Sensitive Information (2022).

2.4. Latest developments

There are several latest developments in OAuth that are being proposed and developed to improve the security and functionality of the protocol by working groups of Internet Engineering Task Force (IETF). These include new proposed standard RFCs, OAuth 2.0 extensions and active internet drafts such as JWT-Secured Authorization Request (JAR) RFC 9101 Sakimura et al. (2021), OAuth 2.0 Pushed Authorization Requests (PAR) RFC 9126 Lodderstedt et al. (2021), Active Internet Drafts such as OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP) Fett et al. (2023) and OAuth 2.1 Hardt, Parecki and Lodderstedt (2022).

2.5. The research impetus

OAuth 2.0 specification differentiates between public and confidential clients and provides different grants for each type of client, which increases the complexity and variability. This also leads to issues with security and scalability as different code bases have to be maintained to cater to different client types.

Additionally, there is no existing unified architecture and approach that addresses known vulnerabilities in OAuth 2.0 Singh and Chaudhary (2022). This can make it difficult for developers to implement OAuth securely and can lead to a lack of consistency across different implementations. Furthermore, the piecemeal implementation and adoption of various standards also produces problems for the interoperability as some of the newer OAuth extensions and RFCs are not backward compatible, which can discourage its widespread adoption. This can make it difficult for different systems and applications to work together seamlessly and limit the potential for secure deployment of the OAuth.

3. Features of USPFO

The unified flow aims to improve the security and consistency of OAuth implementations by introducing new features and best practices. Some of the highlights of the unified flow include:

(1) ***Strengthening the client authentication process:*** A separate client developer or third party controlled endpoint has been introduced to ensure the integrity and authenticity of the client application. It gives flexibility to the developer to incorporate suitable security measures to identify whether the requesting client application is genuine or otherwise. This increases the security of the overall system by providing an additional layer of defense making the system more robust and resistant against token replay and client impersonation attacks. Additionally, this approach allows for easy updating and management of the client application. This approach also follows the principle of separation of duties, as the task of ensuring genuineness of the client application initiating the OAuth process is delegated to a separate endpoint. The authorization server only verifies the client assertion claims through the pre-registered endpoint.

(2) ***Eliminating use of `client_secret`:*** One of the major features of USPFO is to eliminate the use of the `client_secret`, which is traditionally used by confidential clients in the authorization code grant to establish their identity with the authorization server using basic authentication method RFC 7617 Reschke (2015). The `client_secret` is generated by the authorization server at the time of registration of the client application by the developer and is shared with the client. The `client_secret` is sent to the authorization server in base64 encoded form in the authorization code flow grant which makes it susceptible to interception through man-in-the-middle attacks. Once the `client_secret` is compromised, it can be used to impersonate the client, making it a security risk. As the `client_secret` is shared between the client and the authorization server, it becomes difficult to rotate the `client_secret` without updating both the client application and the authorization server and could be a problem as it cannot be easily replaced to prevent its further unauthorized usage CVE-2022-31186 (2022), CVE-2019-5625 (2019), CWE-522: Insufficiently Protected Credentials (2022), CWE-532: Insertion of Sensitive Information into

Log File (2021), CWE-922: Insecure Storage of Sensitive Information (2022).

(3) ***Merging of authorization code grant and the implicit grant:*** This feature eliminates the distinction between the two grants and unifies the architecture and protocol flow, making it easier for developers to implement, mainatin and update OAuth securely.

(4) ***Introducing*** **`assertion_verification_uri`** ***field:*** This new field needs to be registered at the time of registration of the client application with the authorization server. Traditionally, the `client_secret` is used to authenticate the client to the authorization server, but it can be easily leaked or stolen, which can lead to client impersonation. The client assertion verification URI is a developer or third-party provided URI that can be used by the authorization server to retrieve the client's public certificate and use it to verify the signature of the assertion. This also allows for easy management and rotation of keys for the client applications without the need to update the data registered with the authorization server.

(5) ***In-built support for integrity, authenticity, and audience binding:*** These features provide protection against commonly known vulnerabilities such as token leakage and replay attacks. Integrity ensures that the tokens are not tampered with during transit, authenticity ensures that the tokens were issued by a trusted source, and audience binding ensures that the tokens are intended for a specific audience. Together, these features ensure that the tokens exchanged during the OAuth flow are legitimate and cannot be used by an attacker to gain unauthorized access to the protected resources.

(6) ***Authentication of Client before user interaction:*** The Unified Flow (USPFO) enables authentication of client applications before any user interaction takes place as in OAuth 2.0 Pushed Authorization Requests (PAR) RFC 9126 Lodderstedt et al. (2021). This increases the confidence in the client application for subsequent steps in the protocol flow and ensures that the request is coming from a legitimate source before any user interaction happens. This also helps in mitigation of certain kinds of attacks such as denial-of-service as the rogue requests are rejected at the initial stages itself, limiting the allocation and consumption of resources from the authorization server.

(7) ***Compatibility with newer proposed standard RFCs and OAuth extensions:*** The unified flow is largely compatible with many newer proposed standard RFCs and OAuth extensions such as JWT-Secured Authorization Request (JAR) RFC 9101 Sakimura et al. (2021), OAuth 2.0 Pushed Authorization Requests (PAR) RFC 9126 Lodderstedt et al. (2021), Proof Key for Code Exchange by OAuth Public Clients (PKCE) RFC 7636 Sakimura et al. (2015) which improves the security and functionality of the OAuth protocol.

Comparison between traditional OAuth 2.0 flow and USPFO is given at Table 1.

4. The Protocol Flow

The data and protocol flow for USPFO is outlined at Figure 1 and described below.

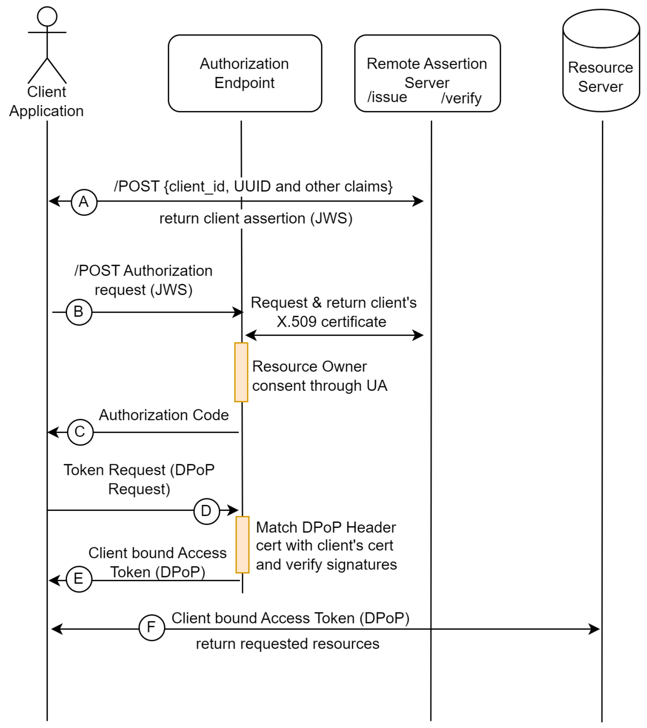

Client Application
Authorization Endpoint
Remote Assertion Server
/issue
/verify
Resource Server
/POST {client_id, UUID and other claims}
A
return client assertion (JWS)
/POST Authorization request (JWS)
B
Request & return client's X.509 certificate
Resource Owner consent through UA
Authorization Code
C
Token Request (DPoP Request)
D
Match DPoP Header cert with client's cert and verify signatures
Client bound Access Token (DPoP)
E
F
Client bound Access Token (DPoP)
return requested resources

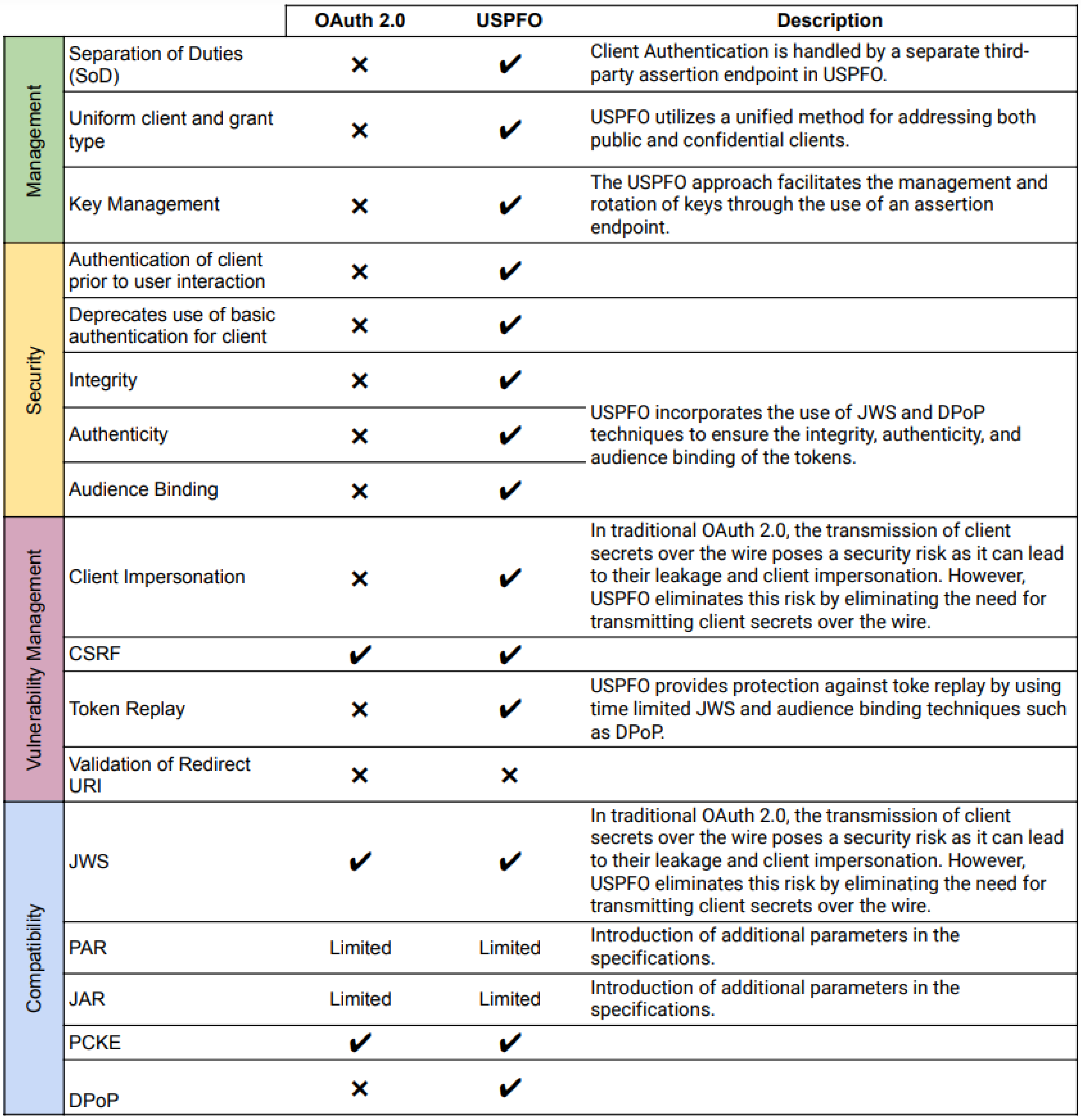
Figure 1: USPFO data and protocol flow

(A) The client application sends a request having client identity (`client_id`) and a random number [version 4 Universally Unique Identifier (UUID) string according to RFC 4122 Leach, Salz and Mealling (2005)] and other claims such as the intended audience to the remote assertion server. The remote assertion server will then return a signed JWT [RFC 7519 Jones, Bradley and Sakimura (2015b)] using the private key associated with the client JWS [RFC 7515 Jones, Bradley and Sakimura (2015a)]. By using a remote assertion server, the client can avoid the need to store private keys on the device and it can also share the same assertion across multiple devices.

(B) The client application sends the authorization request along with the client assertion to the authorization server. The authorization server verifies the authenticity of the client assertion by fetching the client's X.509 certificate from the client assertion verification endpoint, which has been registered with the authorization server at the time of registration of client application by the developer.

(C) If the client assertion is valid, the authorization server will forward the request to the resource owner through a user agent (typically a browser). If the resource owner has granted access, the authorization server will generate an authorization code and send it to the client application via the redirect URI registered with the authorization server.

Table 1
Comparison between OAuth 2.0 and USPFO



		OAuth 2.0	USPFO	Description
Management	Separation of Duties (SoD)	✗	✓	Client Authentication is handled by a separate third-party assertion endpoint in USPFO.
	Uniform client and grant type	✗	✓	USPFO utilizes a unified method for addressing both public and confidential clients.
	Key Management	✗	✓	The USPFO approach facilitates the management and rotation of keys through the use of an assertion endpoint.
Security	Authentication of client prior to user interaction	✗	✓	
	Deprecates use of basic authentication for client	✗	✓	
	Integrity	✗	✓	
	Authenticity	✗	✓	USPFO incorporates the use of JWS and DPoP techniques to ensure the integrity, authenticity, and audience binding of the tokens.
	Audience Binding	✗	✓	
Vulnerability Management	Client Impersonation	✗	✓	In traditional OAuth 2.0, the transmission of client secrets over the wire poses a security risk as it can lead to their leakage and client impersonation. However, USPFO eliminates this risk by eliminating the need for transmitting client secrets over the wire.
	CSRF	✓	✓	
	Token Replay	✗	✓	USPFO provides protection against toke replay by using time limited JWS and audience binding techniques such as DPoP.
	Validation of Redirect URI	✗	✗	
Compatibility	JWS	✓	✓	In traditional OAuth 2.0, the transmission of client secrets over the wire poses a security risk as it can lead to their leakage and client impersonation. However, USPFO eliminates this risk by eliminating the need for transmitting client secrets over the wire.
	PAR	Limited	Limited	Introduction of additional parameters in the specifications.
	JAR	Limited	Limited	Introduction of additional parameters in the specifications.
	PCKE	✓	✓	
	DPoP	✗	✓	

(D) The client application will generate the DPoP JWT having claims as defined in the active internet draft for DPoP Fett et al. (2023). The client application will send the DPoP JWT to the remote assertion server. The remote assertion server will sign the JWT (JWS) and return the same to the client. Additionally, it could also add the public key associated with the client as required by the DPoP specifications. The client will send the DPoP proof JWT in the DPoP HTTP header for the access token request.

(E) The authorization server will fetch the client's X.509 certificate from the remote client assertion verification endpoint and then validate the DPoP proof JWT by checking the signature, the claims, and the public key bound to the client's identity. If the DPoP proof JWT is valid, the authorization server will issue an access token (optionally refresh token) that is bound to the public key included in the DPoP proof JWT. This ensures that the access token can only be used by the client that possesses the private key associated with the public key bound to the client's identity.

(F) After the client has received the access token from the authorization server, it can use the access token to request protected resources from the resource server. The resource server checks the signatures and the validity of the access tokens and if these checks fail or the data in the DPoP proof is wrong, the server refuses to serve the request.

5. Actors and their Interactions

5.1. Client Registration

The client application developer needs to register the client type (confidential or public) and client redirection URI with the authorization server as specified in Section 2 of RFC 6749 Hardt (2012). In the unified flow architecture we introduce an additional client type *unified*.

The addition of the *unified* client type allows the authorization server to distinguish between the conventional and the unified singular flow for OAuth (USPFO). The unified flow (USPFO) requires the authorization server to fetch the client's associated X.509 certificates from a pre-registered endpoint.

The `client_id` generated by the authorization server for the *unified* client type should be prefixed by `UFO_`, which would help in recognising the USPFO client. This would also enable authorization server to know which flow to use when processing the request.

In addition to registering the client type and client redirection URI, the client application developer also needs to register an additional field called `assertion_verification_uri` for the *unified* client type. This field is a reference to a trusted source that contains the client's X.509 certificates. The URI provided should be in compliance with the format specified in RFC 3986 Berners-Lee, Fielding and Masinter (2005), which is the standard for URI (Unified Resource Identifier) syntax. This `assertion_verification_uri` is used by the authorization server to fetch the client's X.509 certificates during the *unified* flow of OAuth 2.0. The client assertion verification endpoint is an important part of the authentication process and helps to ensure the confidentiality and integrity of the client's credentials.

As a backup measure, the authorization server may decide to store valid client's public keys as a JSON object containing a valid JWK Set (JSON Web Key Set) as defined in RFC 7517 Jones (2015). This is to account for situations where the specified URI as referenced in `assertion_verification_uri` is not available or not online. However, using a URI is the preferred method as it allows the client to rotate its certificates without the need to make corresponding changes in the authorization server data. The client can use self-signed certificates or trusted third-party signed certificates for authentication. The self-signed certificate method allows for client authentication without the need to maintain a Public Key Infrastructure (PKI) RFC 5280 Boeyen, Santesson, Polk, Housley, Farrell and Cooper (2008).

5.2. Remote Assertion Server

The use of a remote assertion server allows client applications to avoid the need to store private keys on the device, and to share the same assertion across multiple devices. The remote assertion server generates and signs the client assertion, which is then sent back to the client application, who will include it in the authorization request to the authorization server.

The remote assertion server can also be used to generate client assertions for other OAuth extensions such as JWT-Secured Authorization Request (JAR), OAuth 2.0 Pushed Authorization Requests (PAR) RFC 9126 Lodderstedt et al. (2021) and Demonstrating Proof-of-Possession at the Application Layer (DPoP) Fett et al. (2023).

5.3. Client Assertion Request

The client application generates a JWT having client identity in the form of `client_id`, a rondom number [Universally Unique Identifier (UUID) based 128-bit random number as defined in RFC 4122 Leach et al. (2005)] and other claims such as the intended audience.

An example of a request that a client application would send to the remote assertion server to obtain a signed client assertion is given at Listing 1.

Listing 1: Client assertion request

```
POST /assertion/issue HTTP/1.1
Host: assertion.example.com
Content-Type: application/json

{
 "alg": "RS256",
 "typ": "JWT",
 "aud": "https://server.example.org/token",
 "client_id": "UFO_s6Bk8dRkqt3",
 "jti": "5e5ede50-dc60-40b0-bc94-cb2115ad6820"
}
```

The request body is in JSON format and contains the following claims:
"alg": The signing algorithm used to sign the JWT. "RS256" in this case.
"typ": The type of the token, "JWT" in this case.
"aud": The intended audience of the JWT. The URI of the authorization server's token endpoint in this case.
"client_id": The client_id of the client application.
"jti": A unique identifier for the JWT, a UUID in this case.

On receiving a request from the client application, the remote assertion server will first determine the identity and authenticity of the client application through a predetermined method. This method can vary depending on the implementation and could include techniques such as checking for a valid digital signature or comparing a hash of the application binary to a known good value and is out of scope for the purpose of this specification.

Once the remote assertion server establishes the identity of the client application it will return a signed and Base64Url encoded JWT (JWS) [RFC 7515 Jones et al. (2015a)] as a response to the client application.

The authorization server can use the client's public key to verify the authenticity of the JWT, and thus authenticate the client, before any user interaction happens. This approach allows the remote assertion server to act as an independent third party that can authenticate the client without the need for the authorization server to maintain a copy of the client's secret. This also allows for more flexibility in terms of how the client is authenticated.

An example of the signed and Base64URL encoded JWT using RS256 as defined in JSON Web Signature (JWS) RFC 7515 Jones et al. (2015a) is given at Listing 2.

Listing 2: Signed and Base64URL encoded JWT using RS256

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJhbGciOiJSUzI1Ni
IsInR5cCI6IkpXVCIsImF1ZCI6Imh0dHBzOi8vc2VydmVyLmV4YW1wb
GUub3JnL3Rva2VuIiwiaXNzIjoiVUZPX3M2Qms4ZFJrcXQzIiwic3Vi
IjoiVUZPX3M2Qms4ZFJrcXQzIiwiZXhwIjoxNjIzOTAyMjEzLCJqdGk
iOiIxMjM0NTY3ODkwIn0.Ho4GQlnIzcNcCg5lwYGDihMU07fQTMMzQI
RlANgjadO_OrtD0X7s2w-fm9kXcdIYJ7RpLH-41jfJ6HKOIbgAkvS9D
RvVctUR_cw3yJCC8g1tPagrw3gFDkVMBi5R0sB7q-Tl47zwFhWU2sNE
9b2g4qNWSZSZGTdVzxdzDuEhQoQNS1a2iRjZe8euBqtqfJGPlZJs61T
1peiDcJtprlb0nGs63VX0PSWUbwT8ZouL1qdOGYiZY0UurzARxOrlhB
YDw42x9fxkkZQaKspFI-ixa_Qwm2Y2TN7rRiQ9VPym_iC3H4NGEPfFm
6Xh2Cn_of9rCfMoN8OZSYz1pJ81XkxI4A
```

5.4. Certificate Request

The authorization server requests the X.509 certificate associated with the `client_id` from the remote client assertion verification endpoint. An example of the request message sent by the authorization server to the client assertion verification endpoint is at Listing 3. The request is sent as a HTTP POST to the endpoint's URI, such as `/cert` in this example. The host header is set to the domain of the client assertion verification endpoint, in this case `assertion.example.org`. The Content-Type header is set to `application/x-www-form-urlencoded` and the request body contains the `client_id` of the client, prefixed by `UFO_` as previously described. This request is used to fetch the client's X.509 certificate from the client assertion verification endpoint, which the authorization server uses to verify the client's identity and ensure the authenticity of the request.

Listing 3: Certificate request

```
 POST /cert HTTP/1.1
 Host: assertion.example.org
 Content-Type: application/x-www-form-urlencoded

client_id=UFO_s6Bk8dRkqt3
```

5.5. Authorization Request

Once the client has obtained the signed and Base64URL encoded JWT (client assertion) from the remote assertion server, it can initiate a POST authorization request with the client assertion claim as defined in OAuth 2.0 Pushed Authorization Requests (PAR) RFC 9126 Lodderstedt et al. (2021). The client sends the client assertion claim in the request body, typically as a `x-www-form-urlencoded`, along with other required parameters such as the response_type, scope, and redirect_uri. An example of such a request is at Listing 4.

Listing 4: Certificate request

```
POST /as/ufo HTTP/1.1
 Host: as.example.com
 Content-Type: application/x-www-form-urlencoded

 response_type=code
 &state=af0fijsdlkj
 &client_id=UFO_s6Bk8dRkqt3
 &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
 &code_challenge=
 GjuFoFczD6KdsLNRpqtbv0dOlGGLUNEX6WTRSAnIZFc
 &code_challenge_method=S256
 &scope=read%20write
 &client_assertion_type=
 urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%
 3Ajwt-bearer
 &client_assertion=
 eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJhbGciOiJSUzI1
 NiIsInR5cCI6IkpXVCIsImF1ZCI6Imh0dHBzOi8vc2VydmVyLmV4Y
 W1wbGUub3JnL3Rva2VuIiwiaXNzIjoiVUZPX3M2Qms4ZFJrcXQzIi
 wic3ViIjoiVUZPX3M2Qms4ZFJrcXQzIiwiZXhwIjoxNjIzOTAyMjE
 zLCJqdGkiOiIxMjM0NTY3ODkwIn0.Ho4GQlnIzcNcCg5lwYGDihMU
 07fQTMMzQIRlANgjadO_OrtD0X7s2w-fm9kXcdIYJ7RpLH-41jfJ6
 HKOIbgAkvS9DRvVctUR_cw3yJCC8g1tPagrw3gFDkVMBi5R0sB7q
 -Tl47zwFhWU2sNE9b2g4qNWSZSZGTdVzxdzDuEhQoQNS1a2iRjZe8
 euBqtqfJGPlZJs61T1peiDcJtprlb0nGs63VX0PSWUbwT8ZouL1qd
 OGYiZY0UurzARxOrlhBYDw42x9fxkkZQaKspFI-ixa_Qwm2Y2TN7r
 RiQ9VPym_iC3H4NGEPfFm6Xh2Cn_of9rCfMoN8OZSYz1pJ81XkxI4
 A
```

5.6. Authorization Response

Once the resource owner grants the access, the authorization server issues an authorization code to the client application through redirection URI. An example HTTP response in `x-www-form-urlencoded` is at Listing 5:

Listing 5: Authorization Response

```
 HTTP/1.1 302 Found
Location: https://client.example.org/cb?code=
SplxlQBZeOOYbYSW6xSbIA&state=af0fijsdlkj
```

5.7. Token Request

The client will make a request to the remote assertion server by sending the JSON data as defined in the DPoP (Demonstrating Proof-of-Possession at the Application Layer) Fett et al. (2023) header along with the client identity. The remote assertion server will use the client's private key associated with the `client_id` to sign the DPoP proof JWT and return the signed and Base64URL encoded JWT (JWS)[RFC 7515 Jones et al. (2015a)] as a response to the client application.

An example of the POST request sent by client is at Listing 6.

Listing 6: Token request

```
POST assertion/dpop HTTP/1.1
Host: assertion.example.com
Content-Type: application/json

{
  "typ":"dpop+jwt",
  "alg":"ES256",
  "jti":"f5f254f2-15e1-4a87-9ace-e7a30601df79",
  "htm":"POST",
  "htu":"https://server.example.org/token",
   "client_id": "UFO_s6Bk8dRkqt3"
}
```

The client must provide a valid DPoP proof JWT in a DPoP header when sending an access token request to the authorization server's token endpoint in order to request an access token that is bound to a public key of the client using DPoP. The HTTP request shown in Listing 7 demonstrates such an access token request utilizing an authorization code grant with a DPoP proof JWT in the DPoP header.

Listing 7: Access token request

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
DPoP:
eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0
eSI6IkVDIiwidXNlIjoic2lnIiwiY3J2IjoiUC0yNTYiLCJ4IjoiZW5W
UXVRWjdmN1k4SFFKdWtqamVCaFJHaWY0VzA2TlhDdk5ENWlVams4ayIs
InkiOiJwMXllUndsN3R5YlBwZWoxbDVkNWhSdUg2V252N01RMDZQbmho
UDVrR2s4IiwiYWxnIjoiRVMyNTYifX0.eyJqdGkiOiJmNWYyNTRmMi0x
NWUxLTRhODctOWFjZS1lN2EzMDYwMWRmNzkiLCJodG0iOiJQT1NUIiwi
aHR1IjoiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5vcmcvdG9rZW4iLCJp
YXQiOjE2NzQzODY3Njd9.YZBDPYwgznwajvZrCyJB3Z0ECDeIbA_3su9
HOrSXhE9yXT2-aVpsp5RhQ2VMCArQWFqvfIEfawMz5y90yi97JQ

grant_type=authorization_code
&client_id=UFO_s6Bk8dRkqt3
&code=SplxlQBZeOOYbYSW6xSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
&code_verifier=
     X3ERFdPnKTKk7aUO3_X87SvykRINAXXwKSOlymaqAn
FTb4hjVE4KVzXgeyNbk06SqIC5G4_2zrcoqZ1cF4nz0-
     beYe04iPdDQ79
EmdcOo3Zajo08oQaXi2R5V8Z3Bk5D
```

In OAuth 2.0 [RFC 6749 Hardt (2012)], confidential clients are required to authenticate with the authorization server and are typically issued client passwords at the time of registration of the client application by the developer with the authorization server. These `client_id` and `client_secret` are used to authenticate client application using basic HTTP authentication. However, in USPFO, client passwords must not be used for client authentication. Instead, client authentication is carried out through the use of a public/private key pair and the `assertion_verification_uri`. This eliminates the need to store `client_secret` and eliminates the risk of `client_secret` being compromised due to insecure storage or intercepted over the wire. Additionally, by using public key cryptography, the client can prove possession of the associated private key. This makes it more secure as compared to basic authentication which only verifies the `client_secret`.

To sender-constrain the access token, the authorization server associates the issued access token with the public key from the DPoP proof. This ensures that the access token can only be used by the client that requested it, and not by any other party. An example response is at Listing 8.

Listing 8: Access token response

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
 "access_token": "Kz~8mXK2aElYnzHw-LC-1fBAo.4jLp~
      zsPE_NeO.gxU",
 "token_type": "DPoP",
 "expires_in": 2677,
 "refresh_token": "Q..Zmk92lexi8VnWg2zPW1x-
      tgaG0dIbc3s3EMw_Ni4-k"
}
```

Resource servers must be able to verify that an access token is bound to a public key using DPoP and have the necessary information to confirm this binding. This could be achieved by linking the public key with the token in a way that the protected resource server can access, such as by including the JWK hash in the token directly or through token introspection.

6. Security Considerations

(1) ***Availability of the remote assertion endpoint:*** The availability of remote assertion endpoint is critical to the USPFO. It needs to be available and accessible at all times for issuing and verification of the client assertions. It should also be able to handle high volume traffic in case of popular applications having a wider audience.

(2) ***Theft of Signed Client Assertions:*** The potential for signed client assertions to be stolen during transit from the remote assertion endpoint to the client, via techniques such as man-in-the-middle attacks, poses a risk for client impersonation. To mitigate this risk, it is important to limit the lifespan of these signed assertions and to use secure transport mechanisms such as Transport Layer Security (TLS) during transmission.

(3) ***Data Integrity from client to assertion endpoint:*** To ensure the integrity of the data sent from the client to the remote assertion server, it is recommended to use a

secure TLS connection. Additionally, a shared secret could be exchanged when the client is first authenticated by the remote assertion endpoint, which could then be used to sign the JWTs sent from the client to the assertion endpoint. This would prevent any unauthorized modifications to the data in transit.

7. Conclusion

In this research paper we presented the concept of a Unified Singular Protocol Flow for OAuth (USPFO) Ecosystem. The current OAuth 2.0 framework has multiple client types and grant types, each with their own set of vulnerabilities and configuration complexities. To address these issues, we proposed USPFO which merges different client and grant types into a single, unified protocol flow. This approach not only simplifies the implementation and configuration process but also provides out-of-the-box integrity, authenticity, and audience binding for exchanged codes and tokens, protecting them against theft, replay and impersonation attacks. Additionally, USPFO is largely compatible with IETF proposed standards RFCs, OAuth 2.0 extensions and active internet drafts, such as JAR, PAR, PKCE, JWS, and DPoP. Overall, USPFO offers a solution to improve the security and usability of OAuth 2.0 for both confidential and public clients.

8. Future Work

Future work would involve evaluating the performance and security of USPFO approach, and preparing it as a draft RFC for submission to IEFT.

References

Arshad, E., Benolli, M., Crispo, B., 2022. Practical attacks on login csrf in oauth. Computers & Security 121, 102859.

Berners-Lee, T., Fielding, R.T., Masinter, L.M., 2005. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986. URL: https://www.rfc-editor.org/info/rfc3986, doi:10.17487/RFC3986.

Boeyen, S., Santesson, S., Polk, T., Housley, R., Farrell, S., Cooper, D., 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280. URL: https://www.rfc-editor.org/info/rfc5280, doi:10.17487/RFC5280.

Bugcrowd, 2023. Crowdsourced cybersecurity platform. URL: https://www.bugcrowd.com.

Carnell, J., Sánchez, I.H., 2021. Spring microservices in action. Simon and Schuster.

CVE-2019-5625, 2019. Available from MITRE. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5625.

CVE-2021-43777, 2021. Available from MITRE. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-43777.

CVE-2022-31186, 2022. Available from MITRE. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31186.

CVE-2022-32217, 2022. Available from MITRE. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-32217.

CVE-2022-32227, 2022. Available from MITRE. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-32227.

CVE-2022-36087, 2022. Available from MITRE. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-36087.

CVE-2022-39222, 2022. Available from MITRE. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-39222.

CVE-2022-43693, 2022. Available from MITRE. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-43693.

CVE-2023-22492, 2023. Available from MITRE. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-22492.

CVE-2023-24428, 2023. Available from MITRE. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-24428.

CWE-20: Improper Input Validation, 2022. Available from MITRE. URL: https://cwe.mitre.org/data/definitions/20.html.

CWE-200: Exposure of Sensitive Information to an Unauthorized Actor, 2022. Available from MITRE. URL: https://cwe.mitre.org/data/definitions/200.html.

CWE-312: Cleartext Storage of Sensitive Information, 2022. Available from MITRE. URL: https://cwe.mitre.org/data/definitions/312.html.

CWE-319: Cleartext Transmission of Sensitive Information, 2022. Available from MITRE. URL: https://cwe.mitre.org/data/definitions/319.html.

CWE-352: Cross-Site Request Forgery (CSRF), 2022. Available from MITRE. URL: https://cwe.mitre.org/data/definitions/352.html.

CWE-522: Insufficiently Protected Credentials, 2022. Available from MITRE. URL: https://cwe.mitre.org/data/definitions/522.html.

CWE-532: Insertion of Sensitive Information into Log File, 2021. Available from MITRE. URL: https://cwe.mitre.org/data/definitions/532.html.

CWE-601: URL Redirection to Untrusted Site ('Open Redirect'), 2022. Available from MITRE. URL: https://cwe.mitre.org/data/definitions/601.html.

CWE-613: Insufficient Session Expiration, 2022. Available from MITRE. URL: https://cwe.mitre.org/data/definitions/613.html.

CWE-922: Insecure Storage of Sensitive Information , 2022. Available from MITRE. URL: https://cwe.mitre.org/data/definitions/922.html.

Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., Waite, D., 2023. OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP). Internet-Draft draft-ietf-oauth-dpop-13. Internet Engineering Task Force. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-dpop/13/. work in Progress.

Fett, D., Küsters, R., Schmitz, G., 2016. A comprehensive formal security analysis of oauth 2.0, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. p. 1204–1215. URL: https://doi.org/10.1145/2976749.2978385, doi:10.1145/2976749.2978385.

HackerOne, 2023. Bug bounty platform. URL: https://www.hackerone.com.

Hammer-Lahav, E., 2010. The OAuth 1.0 Protocol. RFC 5849. URL: https://www.rfc-editor.org/info/rfc5849, doi:10.17487/RFC5849.

Hardt, D., 2012. The OAuth 2.0 Authorization Framework. RFC 6749. URL: https://www.rfc-editor.org/info/rfc6749, doi:10.17487/RFC6749.

Hardt, D., Parecki, A., Lodderstedt, T., 2022. The OAuth 2.1 Authorization Framework. Internet-Draft draft-ietf-oauth-v2-1-07. Internet Engineering Task Force. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-v2-1/07/. work in Progress.

Jones, M., 2015. JSON Web Key (JWK). RFC 7517. URL: https://www.rfc-editor.org/info/rfc7517, doi:10.17487/RFC7517.

Jones, M., Bradley, J., Sakimura, N., 2015a. JSON Web Signature (JWS). RFC 7515. URL: https://www.rfc-editor.org/info/rfc7515, doi:10.17487/RFC7515.

Jones, M., Bradley, J., Sakimura, N., 2015b. JSON Web Token (JWT). RFC 7519. URL: https://www.rfc-editor.org/info/rfc7519, doi:10.17487/RFC7519.

Jones, M., Hardt, D., 2012. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750. URL: https://www.rfc-editor.org/info/rfc6750, doi:10.17487/RFC6750.

Leach, P.J., Salz, R., Mealling, M.H., 2005. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122. URL: https://www.rfc-editor.org/info/rfc4122, doi:10.17487/RFC4122.

Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D., Skokan, F., 2021. OAuth 2.0 Pushed Authorization Requests. RFC 9126. URL: https://www.rfc-editor.org/info/rfc9126, doi:10.17487/RFC9126.

Philippaerts, P., Preuveneers, D., Joosen, W., 2022. Oauch: Exploring security compliance in the oauth 2.0 ecosystem, in: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and

Defenses, Association for Computing Machinery, New York, NY, USA. p. 460–481. URL: https://doi.org/10.1145/3545948.3545955, doi:10.1145/3545948.3545955.

Portswigger, 2023. Web Application Security, Testing, & Scanning. URL: https://www.portswigger.net.

Reschke, J., 2015. The 'Basic' HTTP Authentication Scheme. RFC 7617. URL: https://www.rfc-editor.org/info/rfc7617, doi:10.17487/RFC7617.

Richer, J., Sanso, A., 2017. OAuth 2 in action. Simon and Schuster.

Sakimura, N., Bradley, J., Agarwal, N., 2015. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636. URL: https://www.rfc-editor.org/info/rfc7636, doi:10.17487/RFC7636.

Sakimura, N., Bradley, J., Jones, M., 2021. The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR). RFC 9101. URL: https://www.rfc-editor.org/info/rfc9101, doi:10.17487/RFC9101.

Singh, J., Chaudhary, N.K., 2022. Oauth 2.0: Architectural design augmentation for mitigation of common security vulnerabilities. Journal of Information Security and Applications 65, 103091.

Siriwardena, P., 2020. OAuth 2.0 Fundamentals. Apress, Berkeley, CA. pp. 81–101. URL: https://doi.org/10.1007/978-1-4842-2050-4_4, doi:10.1007/978-1-4842-2050-4_4.

Walls, C., 2022. Spring in Action, Sixth Edition. Manning. URL: https://books.google.co.in/books?id=2zVbEAAAQBAJ.

LinkedIn

Jaimandeep Singh is a seasoned cybersecurity expert with over two decades of experience in designing, developing, and implementing IT and cybersecurity solutions. He is currently working for the Government of India in the cybersecurity field. He holds a Bachelor's and Master's degree in Computer Science and is also pursuing PhD in cybersecurity. He is affiliated with the School of Cyber Security & Digital Forensics at the National Forensic Sciences University, India, where he conducts research in the field of cybersecurity.

Naveen Kumar Chaudhary is a Professor and Dean of School of Cyber Security & Digital Forensics, National Forensic Sciences University, India. He has extensive experience of more than 25 years in Cyber Security, e-Governance, Digital Forensics, Network Security & Forensics and Communication Engineering.