mitrelogo-verysmall-white.pngFURL

Sponsor: Department of Veterans Affairs

Contract No.: VA791-P-0042

Project No.: 40144028

**Document No: MTR140262**

**McLean, VA**

Secure RESTful Interface Profile

Security Analysis and Guidance

The views, opinions and/or findings contained in this report are those of The MITRE Corporation and should not be construed as an official government position, policy, or decision, unless designated by other documentation.

This document was prepared for authorized distribution only. It has not been approved for public release.

©2014 The MITRE Corporation. All rights reserved.

M. Russell

July 2014

This page intentionally left blank.

Abstract

The commercial world has rapidly adopted the Representational State Transfer (REST) architectural style, and multiple efforts at the Department of Veterans Affairs (VA) and among its mission partners are actively pursuing RESTful interface implementations. The VA is in need of a framework for securing RESTful interfaces soon, before a large number of services are deployed that will later need to be re-engineered to meet security requirements.

This document provides security analysis and guidance for implementing secure Representational State Transfer (RESTful) interfaces. It presents a security analysis of the OAuth 2.0 and OpenID Connect 1.0 standards for REST security, including descriptions of known attacks and countermeasures. It explains the security implications and design rationale for two prior deliverables, the OAuth and OpenID Connect profiles. It also incorporates an analysis of existing VA and Federal security policies and their potential impacts on a VA implementation of these REST security standards. Finally, it presents a summary of identified issues and recommendations for VA to move towards adoption and implementation of the profiles and accompanying guidance.

This page intentionally left blank.

Table of Contents

[1. Introduction 1](#_Toc394334629)

[1.1. Task Background 1](#_Toc394334630)

[1.2. Scope and Assumptions 2](#_Toc394334631)

[2. Open Security Standards for RESTful Interfaces 3](#_Toc394334632)

[3. REST Security Patterns 4](#_Toc394334633)

[3.1. Client Delegation Pattern 4](#_Toc394334634)

[3.2. Identity Federation (VA as Relying Party) Pattern 7](#_Toc394334635)

[3.3. Identity Federation (VA as OpenID Provider) Pattern 9](#_Toc394334636)

[3.4. Potential Future Pattern – User-Managed Access 11](#_Toc394334637)

[4. Security Analysis 12](#_Toc394334638)

[4.1. Threat Model and Known Attacks 12](#_Toc394334639)

[4.1.1. OAuth 2.0 Threat Model and Known Attacks 12](#_Toc394334640)

[4.1.1.1. Resource Owner Security Issues 12](#_Toc394334641)

[4.1.1.2. Client Security Issues 13](#_Toc394334642)

[4.1.1.3. Authorization Server Issues 14](#_Toc394334643)

[4.1.1.4. Protected Resource Issues 14](#_Toc394334644)

[4.1.1.5. Sample Attack Description – Redirect URI Substitution 15](#_Toc394334645)

[4.1.1.6. Additional Attacks against OAuth 16](#_Toc394334646)

[4.1.2. OpenID Connect Threat Model and Known Attacks 20](#_Toc394334647)

[4.1.2.1. End User Security Issues 21](#_Toc394334648)

[4.1.2.2. Relying Party Security Issues 21](#_Toc394334649)

[4.1.2.3. OpenID Provider Security Issues 22](#_Toc394334650)

[4.1.2.4. UserInfo Endpoint Issues 22](#_Toc394334651)

[4.1.2.5. Attacks against OpenID Connect 22](#_Toc394334652)

[4.2. Rationale for Profiling Decisions 23](#_Toc394334653)

[4.2.1. OAuth Profile 23](#_Toc394334654)

[4.2.1.1. Client Types 24](#_Toc394334655)

[4.2.1.2. Client Authentication 24](#_Toc394334656)

[4.2.1.3. Client Redirect URI Restrictions 25](#_Toc394334657)

[4.2.1.4. Use of State Parameter 26](#_Toc394334658)

[4.2.1.5. Access Token Format 26](#_Toc394334659)

[4.2.1.6. Additional Authorization Server Requirements 27](#_Toc394334660)

[4.2.1.7. Advanced Security Options 28](#_Toc394334661)

[4.2.2. OpenID Connect Profile 29](#_Toc394334662)

[4.2.2.1. ID Token Requirements 29](#_Toc394334663)

[4.2.2.2. UserInfo Requirements 29](#_Toc394334664)

[4.2.2.3. Authorization Request Objects 30](#_Toc394334665)

[4.2.2.4. Authentication Context Claims 30](#_Toc394334666)

[4.3. Security Policy Analysis 31](#_Toc394334667)

[4.3.1. Relevant Policy Documents 31](#_Toc394334668)

[4.3.2. Policy Considerations 31](#_Toc394334669)

[4.3.2.1. OAuth Clients and System Interconnections 31](#_Toc394334670)

[4.3.2.2. Level of Assurance Requirements for Access to Patient Data 32](#_Toc394334671)

[4.3.2.3. Public-key Cryptography without X.509 Certificates 33](#_Toc394334672)

[5. Ongoing Task Team Work 33](#_Toc394334673)

[5.1. Pilot Implementation 33](#_Toc394334674)

[5.2. Ongoing Outreach Efforts 34](#_Toc394334675)

[6. Summary and Guidance on Next Steps 34](#_Toc394334676)

[6.1. Steps towards Adoption of the Standards Profiles 34](#_Toc394334677)

[6.2. Summary of Identified Issues and Recommendations 35](#_Toc394334678)

[List of Acronyms 37](#_Toc394334679)

[References 39](#_Toc394334680)

List of Figures

[Figure 1 - REST Security Standard Dependencies 4](#_Toc394334681)

[Figure 2 – Client Delegation Pattern 5](#_Toc394334682)

[Figure 3 - OAuth 2.0 Authorization Code Flow 6](#_Toc394334683)

[Figure 4 – Identity Federation (VA as RP) Pattern 7](#_Toc394334684)

[Figure 5 – OpenID Connect Authorization Code Flow 8](#_Toc394334685)

[Figure 6 – Identity Federation (VA as OP) Pattern 9](#_Toc394334686)

[Figure 7 – OpenID Connect Authorization Code Flow 10](#_Toc394334687)

[Figure 8 - UMA Architecture 11](#_Toc394334688)

List of Tables

[Table 1 – Open Security Standards for RESTful Interfaces 3](#_Toc394334689)

[Table 2 - Additional Attacks against OAuth 17](#_Toc394334690)

[Table 3 - Attacks against OpenID Connect 22](#_Toc394334691)

[Table 4 - Relevant Policy Documents 31](#_Toc394334692)

# Introduction

This document provides security analysis and guidance for implementing secure Representational State Transfer (RESTful) interfaces for the Department of Veterans Affairs (VA) Deputy Chief Information Officer, Office of Information and Technology (OIT), Architecture, Strategy, and Design (ASD).

## Task Background

The MITRE Corporation’s work program in support of VA ASD is organized by VA outcomes. One of the ASD outcomes is *Integration with Mission Partners*. The Integration with Mission Partners outcome statement is: *The VA interoperates seamlessly with mission partners enabled by architectural alignment*. This Outcome is focused on VA interactions with external entities, including:

* The US Department of Defense (DoD)
* Other Federal agencies
* Health care providers
* Other commercial organizations
* Veterans, Veteran Support Organizations, and Caregivers
* The general public

MITRE’s 2014 support to the Integration with Mission Partners outcome is divided into Task Areas. The *Modern Open Architecture* Task Area, which includes work on Secure RESTful Interfaces, provides technical contributions to accelerate VA’s use of open standards to facilitate mission partner interoperability.

This work is based on the premise that adopting open standards that are in wide use on the commercial web today, such as the RESTful architecture style, will enable VA to integrate with external partners across all of the categories listed above with less expense and effort and with greater reuse of deployed systems and interfaces. Adoption of new architectures such as REST is necessary to support Strategic Objective 3.2 of the VA Strategic Plan: “Evolve VA Information Technology Capabilities to Meet Emerging Customer Service / Empowerment Expectations of Both VA Customers and Employees” [1].

Growing interest in REST is evident at the VA and among its Federal and commercial mission partners, as well as the wider Health Information Technology (IT) community. Given the VA’s responsibility to safeguard information, including the private health information of the Nation’s veterans, it is incumbent on the VA to adopt new technologies in a secure, reliable, and trustworthy manner. MITRE’s work on Secure RESTful Interfaces offers technical guidance in support of this objective. The focus of this MITRE task is not on specific RESTful interfaces themselves, but rather how they can be secured using open security standards.

This paper documents the work conducted through Phase 1 of the Secure RESTful Interface Profile task, along with three companion documents:

* Secure RESTful Interfaces: Business-oriented Use Cases & Associated Distributed Security Requirements [2] – This document introduces the RESTful architecture style and some of its benefits, and identifies RESTful security patterns, sample use cases, and security requirements.
* Secure RESTful Interfaces: Draft Profiles for the Use of OAuth 2.0 [3] – profiles constraining the use of OAuth 2.0 to provide required security controls for VA use
* Secure RESTful Interfaces: Draft Profiles for the Use of OpenID Connect [4] – profiles constraining the use of OpenID Connect 1.0 to provide the required security controls for VA use

This document summarizes the REST security patterns documented [2], such as client delegation and identity federation. It provides a security analysis of the OAuth 2.0 and OpenID Connect 1.0 standards and a discussion of known attacks and countermeasures. It also explains the design decisions made in defining the draft OAuth and OpenID Connect profiles and the rationales for them. Following that is an analysis of current VA and Government security policies pertinent to VA adoption of REST security standards, which identifies some policy considerations. Finally, the document discusses the next steps for the Secure RESTful Interface task and steps the VA can take towards adoption of the guidance and standards profiles produced under this task.

The second phase of the Secure RESTful Interface task will select a pilot use case and produce a working pilot implementation to demonstrate the viability of REST interfaces built using the proposed guidelines and profiles.

## Scope and Assumptions

The following paragraphs state the underlying assumptions and scope boundaries for the Secure RESTful Interfaces work.

1. Focus on External Interfaces

The Secure RESTful Interfaces task is focused on interfaces between the VA and external entities, including the DoD, other Federal agencies, health care providers and other commercial entities, veterans and caregivers, and the general public. Interfaces intended for internal VA consumption are out of scope.

1. Address a Representative Sample of Use Cases

The use cases outlined in the Use Cases document [2] provide a representative sample of use cases for secure RESTful interfaces. It is not a comprehensive list, nor is it meant to constrain the set of use cases to which a secure RESTful approach could be applied.

1. Address Multiple Domains

The scope of this task is not restricted to the health care domain. Given the VA’s mission, health care is clearly a major area of focus, but this task’s goal is to produce security guidance that is applicable across multiple domains.

1. Use Open Standards

A core tenet of ASD’s architecture strategy is the adoption of open standards to maximize interoperability and reduce total cost of ownership of applications. This task recommends the use of suitable established open standards in support of this goal.

1. Provide Interface-Agnostic Guidance

In order for the security guidance produced through this task to be applicable across multiple domains and interfaces, it must necessarily be agnostic to the specific content of any given REST Application Programming Interface (API). At the same time, security guidance must acknowledge that APIs will handle data with varying levels of sensitivity, and different business processes will need to address different risks and threats. Where appropriate, the guidance includes options that may be selected based on the needs of a particular API and implementation.

1. Take a Forward-Looking Approach

The guidance provided will aid VA in moving toward its desired future-state architecture. In addition to well-established standards and technologies that are available today, the guidance produced for this task also considers the potential roles of emerging standards and technologies.

# Open Security Standards for RESTful Interfaces

Table 1 provides a brief description of open security standards defined by the Internet Engineering Task Force (IETF), the OpenID Foundation (OIDF), and other standards organizations for securing RESTful web interfaces. The Use Cases document [2] provides additional details about these standards.

Table – Open Security Standards for RESTful Interfaces

| Standard | Description |
| --- | --- |
| Transport Layer Security (TLS) [5] | IETF standard for secure communications between a client and server, providing transport-layer encryption, integrity protection, and authentication of the server using X.509 certificates (with optional client authentication) |
| OAuth 2.0 [6] | IETF standard for an authorization framework whereby resource owners can authorize delegated access by third-party clients to protected resources; OAuth enables access delegation without sharing resource owner credentials, with optional limits to the scope and duration of access |
| JavaScript Object Notation (JSON) [7] | Ecma[[1]](#footnote-1) standard text format for structured data interchange – not a security standard per se, but a key component of several standards listed here |
| JSON Web Signature (JWS) [8] | Draft IETF standard for attaching digital signatures or Message Authentication Codes (MAC) to JSON objects |
| JSON Web Encryption (JWE) [9] | Draft IETF standard for encrypted JSON objects |
| JSON Web Keys (JWK) [10] | Draft IETF standard for representing public and private keys (or sets of keys) as JSON objects |
| JSON Web Algorithms (JWA) [11] | Specifies cryptographic algorithms to be used in the other JOSE standards |
| JavaScript Object Signing and Encryption (JOSE) | Collective name for the set of JSON-based cryptographic standards (JWS, JWE, JWK, and JWA) |
| JSON Web Token (JWT) [12] | Draft IETF standard for conveying a set of claims between two parties in a JSON object, with optional signature and encryption provided by the JOSE standards |
| OpenID Connect 1.0 [13] | OpenID Foundation standard for identity federation based on OAuth 2.0, using JWT to convey signed and optionally encrypted identity claims |
| User-Managed Access (UMA) [14] | Draft IETF standard for an OAuth 2.0-based access management protocol enabling resource owners to create access policies authorizing requesting parties to access their resources through OAuth clients |

Figure 1 below illustrates the dependencies among the security standards, with each standard depending on the others that lie directly beneath it.

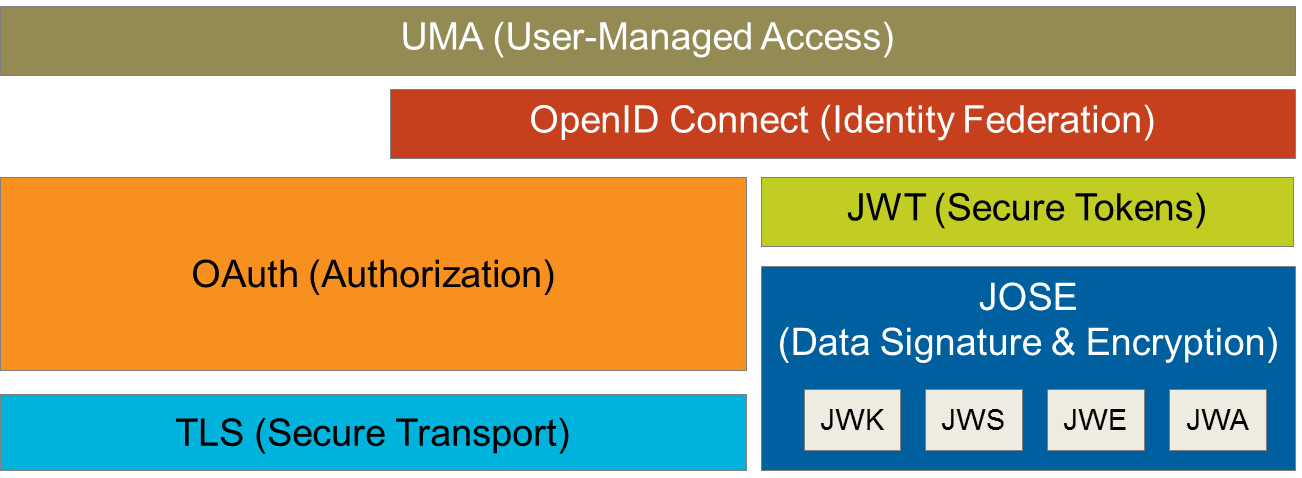


Figure - REST Security Standard Dependencies

# REST Security Patterns

In order to provide general guidance on REST security, the Secure RESTful Interfaces Use Cases document [2] is organized by a set of general REST security patterns that can apply to a number of different use cases. This set of use case patterns was the basis for developing the OAuth and OpenID Connect standards profiles and the guidance in this document. This document briefly summarizes the security patterns; but more detail is available in the Use Cases document.

## Client Delegation Pattern

This pattern enables a Resource Owner to grant a third-party software client delegated access to a protected resource – the canonical use case for which OAuth was created. In this pattern, the protected resource resides on a VA system.

Figure 2 below provides a functional depiction of this use case. This figure does not represent the actual protocol message flow among participants, since this would depend on the specific OAuth flow used (Authorization Code, Implicit, etc.).

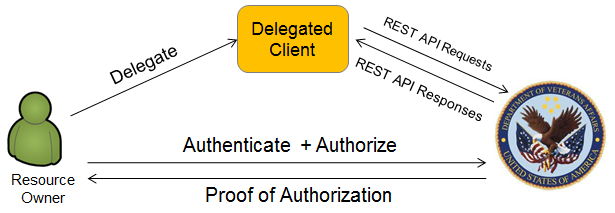


Figure – Client Delegation Pattern

Specific examples of the Client Delegation pattern include:

* A veteran delegates access to his/her Electronic Health Record (EHR) to a web application for tracking blood pressure
* A veteran delegates access to his/her EHR to a mobile application, limiting the scope of access to information pertaining to a particular medical condition
* A veteran uses a personal health monitoring device which uploads measurement data to an external web application; the veteran delegates access to the web service to upload this Patient-Generated Data (PGD) to a VA system

In any of these cases, the Resource Owner could instead be a Veteran Service Organization (VSO) or other authorized caregiver.

The following sequence is based on the OAuth Authorization Code flow, depicted in Figure 3. Using the use case example involving a veteran delegating EHR access to a web application for tracking blood pressure, the italicized text in the diagram indicates which role each of these parties would play in the protocol flow. The “user agent” referenced in the following description is a software client, commonly a web browser, that is under the direct control of the Resource Owner. The “Client” in OAuth terms is also a piece of software that performs some action on the Resource Owner’s behalf, but it may or may not be under the Resource Owner’s control. Clients may be native applications, hosted web applications, or applications that run directly in a web browser.

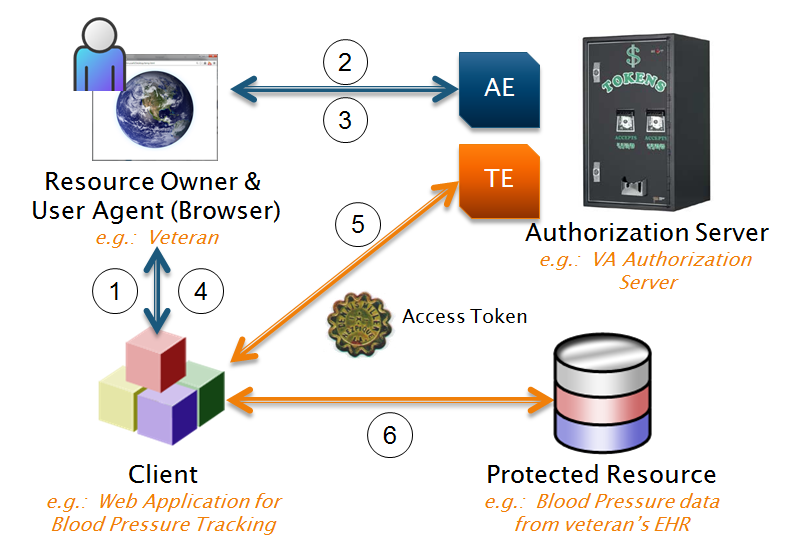


Figure - OAuth 2.0 Authorization Code Flow

1. The Resource Owner interacts with the Client and initiates a request to access the Protected Resource. The details of this process are client-specific.
2. The client redirects the Resource Owner’s user agent to the Authorization Server’s Authorization Endpoint (AE), which prompts the Resource Owner to authenticate.
3. The Authorization Server prompts the Resource Owner to authorize the Client’s access to the Protected Resource, displaying available information about the request, including the requested scope of access.
4. The Authorization Server redirects the Resource Owner’s user agent back to the Client. If the resource owner has approved the request, the redirected message contains an Authorization Code; if the request was denied, it contains an error message.
5. The Client connects to the Authorization Server’s Token Endpoint (TE), using its client credentials to authenticate. The Client submits the Authorization Code, which the token endpoint validates. If the code is valid, the Token Endpoint issues an access token, and optionally a Refresh Token.
6. The Client connects to the Resource Server, requests access to the Protected Resource, and presents the Access Token as proof of authorization.

In addition to the Authorization Code flow shown above, OAuth defines the following alternative flows:

* Implicit flow – this flow is intended for browser-embedded clients. Because there is no separation between the client and the resource owner’s user agent, the authorization server returns an access token directly from the authorization endpoint, instead of issuing an authorization code and requiring the client to make an additional connection to the token endpoint.
* Client credentials flow – this flow is intended for clients that do not act on behalf of a particular resource owner (for example, clients that perform bulk transfers between systems). Since no resource owner is involved, the client simply requests a token from the token endpoint, authenticating with its own client credentials, and the authorization server issues an access token.
* Resource owner credentials flow – this flow is intended for legacy clients in transition to using OAuth. In this flow, the resource owner provides his/her credentials to the client. The client submits a request to the token endpoint, authenticating with its client credentials and submitting the resource owner’s credentials as a stand-in for the authorization code. The token endpoint returns an access token.

## Identity Federation (VA as Relying Party) Pattern

In this pattern, a VA system, the Relying Party (RP), accepts authenticated identity information from an external OpenID Provider (OP) as a basis for granting a user access to VA resources. The exchange of identity information among trusted partners is called Identity Federation. VA’s current use of DoD Self-Service Logon (DS Logon) credentials managed by the DoD is another example of Identity Federation, though that integration is based on the Security Assertion Markup Language (SAML) rather than OpenID Connect.

The high-level scenario for this pattern is shown in Figure 4. An external user authenticates to an external OP, which then provides an ID Token to a VA system. The ID Token conveys an assertion that the user identified in the token has been authenticated by the OP. The OP also provides the VA with an Access Token, which can be submitted to the OP’s UserInfo endpoint in order to obtain additional claims, or attributes, about the user.



Figure – Identity Federation (VA as RP) Pattern

Specific examples of the Identity Federation (VA as RP) pattern include:

* A VSO employee authenticates to a VA system through an OP operated by the VSO organization
* A specialist authenticates to a VA system through an OP operated by the external health care provider organization in order to view the EHR of a referred patient
* A pharmacy employee authenticates to a VA system through an OP operated by the pharmacy to upload veteran immunization records
* Veterans authenticate through the OP of their choice as an alternative to DS Logon

Like OAuth, OpenID Connect supports multiple protocol flows. The sequence depicted in Figure 5 is based on the Authorization Code flow. This flow sequence is described below. Using the use case example involving a VSO employee authenticating to a VA application through a VSO-operated OP, the italicized text in the diagram indicates which role each of these parties would play in the protocol flow.

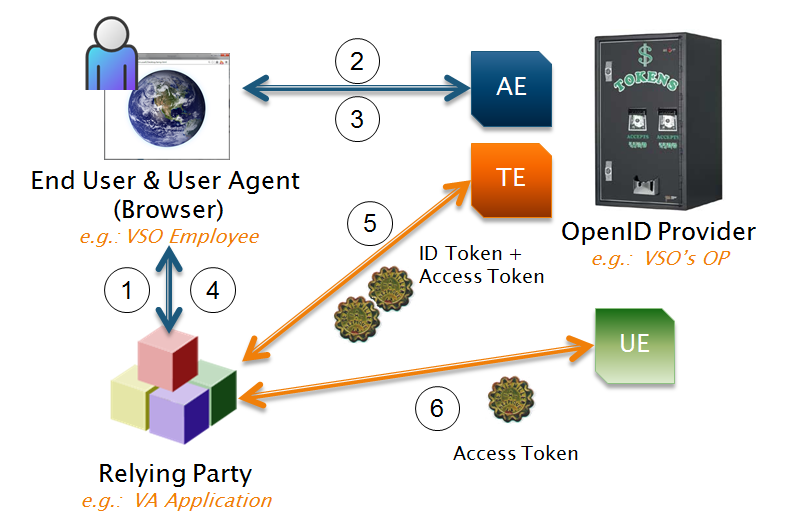


Figure – OpenID Connect Authorization Code Flow

1. An external (non-VA) End User attempts to access a VA-hosted application, the RP. The RP prompts the end-user to authenticate. The RP discovers that the user has an account with the external OP (e.g., by providing a “Sign in with your <OP Name> account” link which the user clicks).
2. The RP redirects the End User’s user agent to the OP’s Authorization Endpoint (AE) with an Authentication Request. In this pattern, the OP is operated by an organization external to the VA.
3. The OP prompts the End User to authenticate, and to authorize the VA RP to access claims about his/her identity (e.g., e-mail or postal address, telephone numbers, organizational role).
4. If the End User authenticates successfully and authorizes access, the OP redirects the End User’s user agent back to the RP with an Authorization Code.
5. The RP submits its Authorization Code to the OP’s Token Endpoint (TE), and authenticates with its client credentials. If the OP successfully validates the Authorization Code and client authentication succeeds, the OP responds with both an ID Token and an Access Token.
6. The RP validates the ID Token and extracts the End User’s subject identifier (from the “sub” claim). Optionally, the RP may request additional claims about the End User from the OP’s UserInfo Endpoint (UE), presenting the Access Token for authorization.

In addition to the Authorization Code flow shown above, OpenID Connect defines the following alternative flows:

* Implicit flow – similar to the OAuth implicit flow (and also intended for browser-embedded clients) – the authorization endpoint issues an ID Token (and an Access Token, if requested) without involvement of the Token Endpoint
* Hybrid flows - the Authorization Endpoint returns an Authorization Code along with an ID Token, an Access Token for the UserInfo Endpoint, or both. As with the OAuth implicit flow, the hybrid flows expose tokens to the end user’s browser.

## Identity Federation (VA as OpenID Provider) Pattern

This pattern is essentially the same as the previous Identity Federation pattern but with a reversal of roles. In this pattern, VA provides authenticated identity information to an external RP application, enabling VA users to access it with their existing credentials. While the previous pattern required the VA to establish a trust relationship with an OP to provide authenticated user information, in this case the VA would trust the external party with access to information about its own users. The benefit to the VA’s users would be access to external applications without the need to establish new credentials.

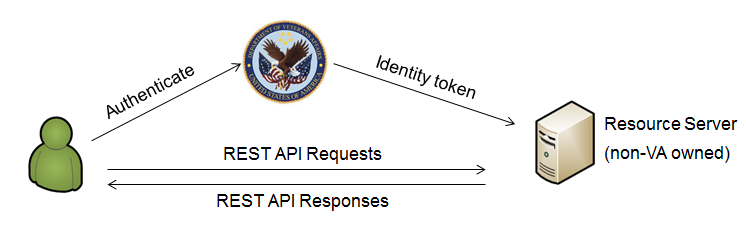


Figure – Identity Federation (VA as OP) Pattern

Specific examples of the Client Delegation pattern include:

* A VA physician accesses an external health care provider’s web portal to access records of a veteran’s referred consultation with a specialist
* A VA cybersecurity analyst accesses an external Poll service to obtain cyber threat information using a service such as Trusted Automated Exchange of Indicator Information (TAXII) [15]

The following sequence is based on the Authorization Code flow. The protocol flow, depicted in Figure 7, is the same as the one presented in section 3.2, apart from the reversal of roles between the VA and the external organization. The italicized text references the use case example involving a VA user authenticating to an external health care provider’s web portal through a VA-operated OP.

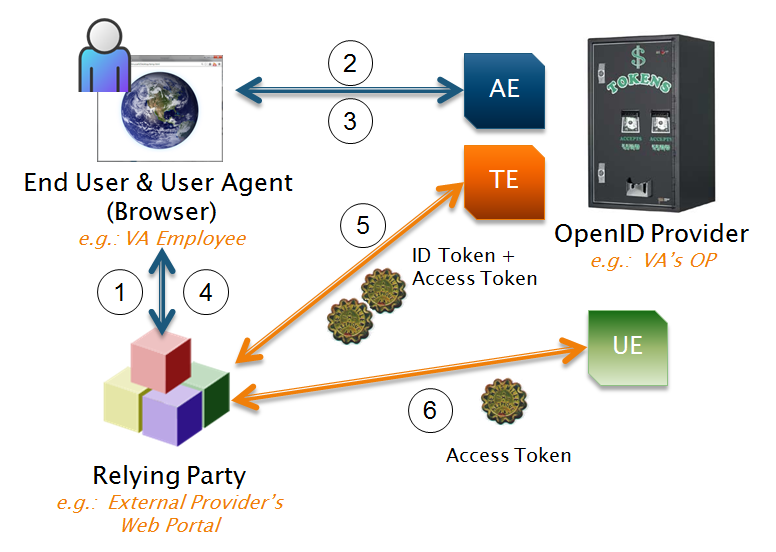


Figure – OpenID Connect Authorization Code Flow

1. The End User attempts to access the external RP; the RP prompts the end-user to authenticate. The RP discovers that the user has an account with the VA OP (e.g., by providing a “Sign in with your VA account” link which the user clicks).
2. The RP redirects the End User’s user agent to the VA OP with an Authentication Request.
3. The OP prompts the End User to authenticate, and to authorize the RP to access claims about his/her identity (e.g., e-mail or postal address, telephone numbers, organizational role).
4. If the End User authenticates successfully and authorizes access, the OP redirects the End User’s user agent back to the RP with an Authorization Code.
5. The RP submits its Authorization Code to the OP, and authenticates with its client credentials. If the OP successfully validates the Authorization Code and client authentication succeeds, the OP responds with both an ID Token and an Access Token.
6. The RP validates the ID Token and extracts the End User’s subject identifier (from the “sub” claim). Optionally, the RP may request additional claims about the End User from the OP’s UserInfo Endpoint (UE), presenting the Access Token for authorization.

## Potential Future Pattern – User-Managed Access

While the OAuth 2.0 Framework is now well-established, this pattern is based on the draft UMA standard [14], which is still evolving. A few real-world UMA deployments exist and some software implementations are available. Also, UMA may see wider adoption over the next few years.

UMA is an OAuth 2.0-based access management protocol. To the standard OAuth set of players including a Resource Owner, Client, Authorization Server, and Protected Resource, UMA adds a Requesting Party, enabling the resource owner to create policies authorizing specific individuals to access resources through a given client. The UMA Architecture is depicted in Figure 8.

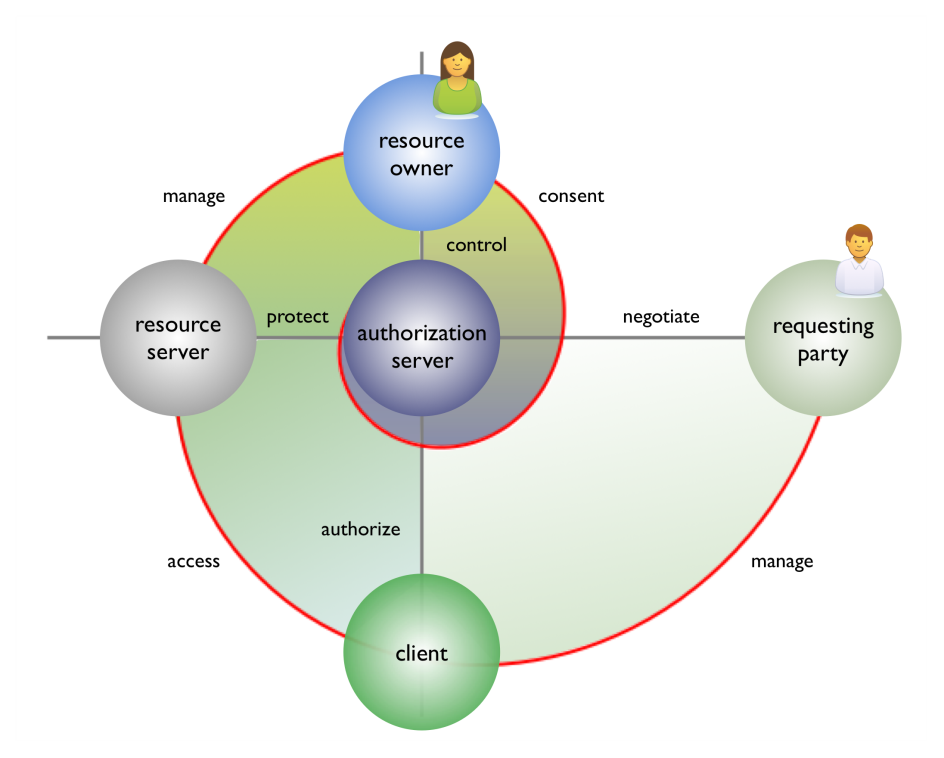


Figure - UMA Architecture [[2]](#footnote-2)

UMA introduces the ability for the Resource Owner to create policies that the authorization server can use to make automated authorization decisions based on specified criteria, rather than requiring the owner to explicitly approve each authorization request. Examples include requiring the requesting party to have certain identity attributes, to have a stated privacy policy meeting requirements specified by the owner, or to agree to non-disclosure terms. Of course, some of these criteria would require self-enforcement by the requesting party, but they could be backed up by contractual agreements among the parties to a given exchange.

The adoption of UMA for VA use cases may not be practical in the near term, but it is presented here as an example of the potential future use cases that could be enabled by RESTful APIs, OAuth, and related standards.

# Security Analysis

## Threat Model and Known Attacks

This section is based on security guidance in the OAuth and OpenID Connect standards documents (notably RFC6819 [16]) and known attacks on implementations. It summarizes the threats and security considerations facing implementations of OAuth and related standards. As stated in the Secure RESTful Interface profiles for OAuth and OpenID Connect created as part of this task, client developers and service providers should read the security guidance in the standards and apply all applicable recommendations.

## OAuth 2.0 Threat Model and Known Attacks

An analysis of OAuth security must consider the interests and interactions of the four principal parties in an OAuth delegation scenario: the resource owner, the client, the authorization server, and the protected resource. The authorization server and the protected resource are frequently, but not necessarily, controlled by the same organization.

### Resource Owner Security Issues

The resource owner has credentials to authenticate to the authorization server, and controls access to some set of protected resources. In deciding to delegate some scope of access to the client, the owner has opted to trust the client (and its developer) to some degree. The resource owner has a vested interest in the security of the protected resources, which in many VA use cases will include access to protected health information.

Potential threats to the resource owner include:

* Interception of resource owner credentials, enabling impersonation of the resource owner to the authorization server
* Delegation of excessive or unintended access to clients
* Malicious or compromised clients abusing delegated access to steal or manipulate data

While the implementation of the authorization server and the client are generally beyond the resource owner’s control, the resource owner performs important security functions in an OAuth implementation. The resource owner evaluates the trustworthiness of clients, makes access control decisions, and uses browser security mechanisms to validate the authenticated Hypertext Transfer Protocol (HTTP) Secure (HTTPS) connection to the authorization server.

Most attacks involve an unauthorized user’s attempt to gain access to a protected resource belonging to an authorized resource owner. However, other scenarios involve resource owners themselves attacking other users. For example, a protected resource might be an API for applying for loans or benefits. An attacker could create a new loan application and trick another user into accessing the application and populating it with sensitive personal data. As the resource owner, the attacker would be able to access the application and retrieve the data submitted by the victim. Another reason for an attacker to direct another user to his protected resources would be to return modified data or malicious content from the legitimate user’s requests. In addition to defending against unauthorized attackers, a secure OAuth implementation must also prevent resource owners from attacking others.

### Client Security Issues

Client security considerations vary among the different client types. Full clients and direct access clients have their own client credentials to protect, but browser-embedded clients do not. Full clients may be native applications installed on many devices, with each instance having its own credentials, or they may be large hosted web applications used by millions of people. A single client compromise may therefore impact a single user, or all of the users of a large website. Through access to protected resources, clients may obtain and store sensitive data.

Malicious clients pose significant risks. They could facilitate phishing attacks by directing resource owners to fraudulent authorization servers in order to collect credentials, leak access tokens to attackers, or simply use their delegated authorities to steal or manipulate data. Clients must register with the authorization server and be issued client IDs (and credentials, except for browser-embedded clients) before they can be used. This enables the organization hosting the authorization server to mitigate some of the risks of malicious clients by vetting and testing them before allowing them to register.

The resource owner can mitigate the risks of malicious clients to some degree, for example by installing signed apps from reputable app markets, and consulting ratings and reviews. Clients depend on the trust of resource owners, and overtly malicious client activity can be reported. Native clients may be removed from app stores, and the owners of web application clients might face legal action as a result of malicious activity. However, activity must first occur in order to be reported, so this approach assumes that at least some resources owners will be victims. Determining the trustworthiness of an OAuth client, or indeed any type of software, is a difficult problem, and it is impossible to eliminate the risks of malicious clients entirely.

Setting aside malicious clients, well-intentioned clients are responsible for several critical security functions such as protecting client credentials (if any), authorization codes, access and refresh tokens, and any data received from the protected resource. The client also orchestrates communications between the resource owner and the authorization server, controlling the URIs to which the owner’s browser is redirected for authorization and to which the authorization server will redirect the user following authorization. Potential threats to clients include:

* Interception of communications between the client and the authorization server to obtain client credentials, authorization codes, or tokens
* Interception of communications between the client and the protected resource to obtain sensitive data
* Authorization code or access token substitution attacks, in which an attacker obtains authorization to the resource owner’s protected resource, or associates the resource owner’s session with the attacker’s own protected resource, through the legitimate client
* Abuse of an open redirect on the client to expose codes or tokens to an attacker-controlled website

### Authorization Server Issues

Responsible for authenticating resource owners and clients and issuing access tokens, the authorization server is a critical component in an OAuth implementation. The OAuth provider which operates the authorization server dictates the rules governing interactions with clients, resource owners, and protected resources. The authorization server controls client registration, enabling the provider to specify vetting and registration metadata requirements (e.g., requiring clients to register their redirect URIs). The provider can dictate client and resource owner authentication requirements, and the structure and format of tokens and codes. In conjunction with protected resource providers, the OAuth provider defines the scopes of access that can be requested and authorized. The correct implementation and management of the authorization server is critical to the security of an OAuth implementation.

Potential threats to the authorization server include:

* Interception of client or resource owner credentials, authorization codes, or access or refresh tokens; attempted replay of intercepted authorization codes
* Brute-force guessing attempts against client or resource owner secrets, or access or refresh tokens
* Manipulation of redirect URIs to lead authorization codes (or access tokens in the implicit flow) to be sent to unauthorized parties
* Cross-site request forgery attacks abusing authenticated resource owner sessions to authorize attacker-controlled clients
* Denial of service attacks

### Protected Resource Issues

The goal of OAuth is to control authorization of access to the protected resource. In the context of REST security, the resource is the RESTful API, which could provide access to any kind of data or functionality. The impact of unauthorized access to the protected resource depends on the specifics of the API, but it could entail theft, deletion, or corruption of data, fraudulent requests for goods or services, or denial of service to authorized users. In the VA’s case, if the API provides access to veterans’ personal data and protected health information, veterans could be the victims of identity fraud, privacy invasion, and a number of other impacts from the unauthorized release of health data.

The protected resource is responsible for accepting and validating access tokens from clients, and granting or denying access to resources as appropriate. At a minimum, the protected resource must determine whether the token is valid and whether it authorizes access to the resource being requested. OAuth does not require a specific mechanism for performing this critical piece of the authorization process, leaving it up to the implementation. The specification does identify options such as the use of signed tokens and token introspection to enable protected resources to validate tokens and obtain associated metadata, including the scope granted by the resource owner.

Potential threats to the protected resource include:

* Presentation of intercepted authorization codes by unauthorized clients
* Presentation of valid access tokens that grant authorization to different protected resources
* Replay of authorized requests
* Requests for resources not included in the authorized scope of access
* Brute-force guessing of access tokens

### Sample Attack Description – Redirect URI Substitution

This technique grants an attacker unauthorized access to a resource owner’s protected resources, using the authorization code flow. Note that this attack will be prevented by security measures specified in the Profiles for the Use of OAuth 2.0 [3].

The sequence of the authorization code flow is described in Section 3.1, but a high-level summary is as follows:

1. The resource owner initiates a request through the client to access a protected resource
2. The client redirects the owner’s user agent to the authorization server, specifying a redirect Uniform Resource Identifier (URI) to which the owner’s user agent should be sent to convey the result of the authorization request back to the client
3. The resource owner authenticates to the authorization server and either approves or rejects the authorization request
4. The authorization server redirects the owner’s user agent to the redirect URI with an authorization code if authorization was granted, or an error otherwise
5. If authorization was granted, the client makes a request to the authorization server’s token endpoint to exchange the authorization code for an access token
6. The client presents the access token to the resource server and is granted access to the resource owner’s protected resources

To conduct this attack, the attacker uses a legitimate web application client to initiate a request for access to a protected resource. Instead of following the redirect to the authorization server in step 2, however, the attacker makes note of the Uniform Resource Locator (URL), but does not access it. The URL of a sample authorization request is shown below:

https://authsvr.example.com/authorize?response\_type=code&client\_id=client1&state=xyz

&redirect\_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb

The *redirect\_uri* parameter contains an encoded URL belonging to the client application, to which the owner’s browser is to be redirected with the results of the authorization request. The attacker modifies this parameter to point to the URL of a site he controls, client.attacker.com:

https://authsvr.example.com/authorize?response\_type=code&client\_id=client1&state=xyz

&redirect\_uri=https%3A%2F%2Fclient%2Eattacker%2Ecom%2Fcb

Next, the attacker somehow entices a resource owner, the victim, to visit this modified link (e.g., by e-mailing it to the victim). The link leads to the authorization endpoint, which will prompt the victim to authenticate and to authorize the client (which is a legitimate application) to access protected resources. The e-mail could include a plausible motivator for the victim to do this, such as a limited-time promotion for users who sign up to use the client application. The victim will recognize the authorization server, where he has an existing account, and it will present a valid TLS certificate, which may increase the victim’s confidence. The e-mail pretext can explain the need to authorize the app. Except for the fact that the sequence begins at the authorization server instead of the client, the OAuth flow appears completely normal to the victim.

If the victim authorizes the request, the authorization server will redirect the victim’s browser to the attacker-controlled site, which may attempt to impersonate the actual client, or may simply return an error message. The attacker needs nothing further from the victim, because the redirected request to the attacker’s server includes the authorization code. The attacker composes a request to the legitimate client and inserts the authorization code received from the victim.

When the client receives the request, it extracts the authorization code and submits it to the authorization server’s token endpoint. Because the attacker is using a legitimate client, it has credentials to authenticate to the authorization server and can exchange the valid authorization code for an access token. The access token is associated with the victim’s protected resources, which the attacker can now access through the client.

This attack, and a number of variations on it that involve manipulation of the redirect\_uri parameter, can be prevented by requiring clients to declare their redirect\_uri values with the authorization server at registration time. The authorization server can then validate the redirect\_uri values on both authorization and token requests against the set of URIs registered for the client to detect manipulation of the parameter. The manipulated redirect\_uri submitted with the authorization request would not match the legitimate client’s registered URIs.

This specific attack can also be detected by comparing the redirect URI submitted in the authorization request with the redirect URI submitted in the corresponding request to the token endpoint. The attacker needs to modify the URI in the authorization request, but since the client sends the request to the token endpoint, the attacker cannot manipulate this message. The client will either include its legitimate redirect URI or omit the parameter entirely. Either way, the authorization server will reject the request because the parameter values are different.

The above example uses a web application client, but the attack is also possible with native application clients. Redirect URI validation can detect the attack with both web and native clients if the attacker uses an HTTP or HTTPS redirect URI to obtain the code.

### Additional Attacks against OAuth

Table 2 below summarizes a number of attacks on OAuth, most of which are described in OAuth Threat Model and Security Considerations [16], and countermeasures that can defend against them. The “Against” column indicates the entity or component that is directly targeted by the attack.

Table - Additional Attacks against OAuth

| Attack | Against | Details | Countermeasures |
| --- | --- | --- | --- |
| Authorization Server spoofing | Resource owner | Resource owner is tricked into authenticating to an attacker-controlled site that impersonates the authorization endpoint to capture credentials, either by a malicious client, or by network address spoofing techniques (e.g., DNS or ARP cache poisoning[[3]](#footnote-3)) | TLS server authentication |
| Clickjacking to obtain authorization | Resource owner | Malicious web site opens the authorization server in an inline frame, hiding its user interface elements behind fake buttons, links, etc. to fool the resource owner into authorizing an access request | * X-FRAME-OPTIONS header * JavaScript frame avoidance methods to support older browsers |
| CSRF[[4]](#footnote-4) to obtain authorization | Resource owner | Malicious web site submits requests to the authorization server, abusing an existing authenticated session in the same browser in order to authorize an access request | CSRF tokens incorporated into the authorization approval form |
| Authentication Server user interface spoofing | Resource owner | When initiating an authorization request, a malicious client displays a user interface that resembles the legitimate authorization endpoint inside an embedded browser or other application interface element, instead of using an external browser, in order to capture resource owner credentials | Require clients to invoke an external browser when directing the resource owner to the authorization endpoint |
| Requesting excessive or inappropriate scope | Resource owner | Malicious client requests access scopes that are unnecessary or inappropriate for the its functionality, which the resource owner may inadvertently grant | * Define scopes with appropriate levels of granularity * Provide clear indications to resource owners what the requested scope entails at the authorization endpoint |
| Redirect URI manipulation | Resource owner, client, authorization server | Several variations:   1. Entice victim to access a modified redirection URL to the authorization server, with a modified redirect\_uri parameter to send the authorization code to an attacker-controlled site 2. If the client has an open redirector vulnerability, modify the redirect\_uri to point to the redirection URL with a parameter to send the authorization code to an attacker-controlled site   When the implicit flow is used, the attacker obtains an access token instead of an authorization code | * Require clients to fully register redirect URIs * Validate submitted redirect URI values * Ensure that clients do not provide open redirectors, particularly through registered redirect URIs * In the authorization code flow, ensure that the redirect URI values in the authorization request and the token endpoint request are the same |
| Interception of authorization code | Authorization endpoint, resource owner | Attacker obtains authorization code (e.g., from captured network traffic or by spoofing a web application client’s address) and exchanges it for an access token to gain access to protected resources | * TLS encryption and server authentication * Prohibit clients from registering redirect URIs using plaintext HTTP * Short validity period for authorization codes * Require client authentication to the token endpoint (ineffective if the attacker can easily register a client) * Detect authorization code replay and invalidate any associated tokens |
| Interception of access or refresh token | Client, token endpoint, protected resource | Attacker obtains access token (e.g., from captured network traffic or inadequate protection of stored tokens on the client or authorization server, or by spoofing the resource server’s network address) and uses it to access protected resources; or obtains refresh token, uses it to obtain an access token, etc. | * TLS encryption * Encryption and/or access control for stored tokens * Associate tokens with the client ID to which they were issued, and ensure that the token is presented by the authorized client [[5]](#footnote-5) * Future – use a proof of possession token mechanism [17] instead of bearer tokens |
| Brute-force guessing of client credentials | Token endpoint | Attacker submits repeated requests to the token endpoint, trying different values for the client ID and secret in order to guess client credentials. Note that this is only effective if the attacker also obtains a valid authorization code. | * Implement client account lockout after failed login count meets a certain threshold within a certain time period, though this may not be acceptable for clients shared by multiple users (e.g., web applications) * Use strong (cryptographic) client authentication |
| Denial of Service using manufactured authorization codes | Token endpoint | An attacker who controls multiple compromised machines (a “botnet”) makes connections to multiple client redirect URIs that use HTTP and submits invalid authorization codes. For each HTTP connection the attacker makes to a client, the authorization server incurs establishment of an HTTPS connection plus whatever processing is required to validate the authorization code and respond to the client. The IP addresses of the machines conducting the attack are hidden from the authorization server, since it only communicates with the clients. | * Client should implement CSRF protection and validate the state parameter on requests to the redirect URI before submitting a request to the token endpoint, though this only requires the attacker to make one additional request to obtain a valid state value and CSRF token * If the client authenticates users, it can lock out clients after meeting a threshold of invalid authorization codes * The authorization server can impose progressive delays in responding to clients who reach a threshold of invalid code submissions, though this may not be acceptable for shared clients (e.g., web applications) |
| Authorization code hijacking | Resource owner, authorization endpoint, token endpoint | 1. Attacker initiates a protected resource request from a legitimate client 2. Attacker modifies the redirect\_uri in the authorization request to point to an attacker-controlled server and entices the victim to access the modified link, authenticate, and authorize the request 3. The victim is redirected to the attacker’s server with the authorization code 4. The attacker submits the authorization code to the real client’s redirect URI 5. The client obtains an access token, enabling the attacker to access the victim’s protected resources through the client | * Require clients to fully register redirect URIs * Validate submitted redirect URI values * Validate that the redirect\_uri parameter values passed to the authorization and token endpoints are the same |
| Authorization code fixation | Client, resource owner | 1. An attacker initiates a protected resource request using a legitimate client to his own protected resource 2. The attacker authenticates to the authorization server and obtains an authorization code 3. The attacker is redirected to the client’s redirect URI, but instead of accessing the URI, he entices the victim to access it 4. The victim follows the link to the client’s redirect URI; the client accepts the authorization code and exchanges it for an access token 5. The victim’s session with the client is now associated with the attacker’s public resource. The victim uploads sensitive data to the attacker’s protected resource through pretext or CSRF | Use the *state* parameter to tie authorization codes to client sessions. Clients should submit an unpredictable *state* value in the authorization request, and associate the *state* value with the client session in which the request was issued (in this case, the attacker’s client session). The authorization endpoint will include the same *state* parameter value in the callback request to the client’s redirect URI, along with the authorization code. This enables the client to validate that the authorization request and the submission of the authorization code occur within the same client session.    When the victim submits the authorization code to the client’s redirect URI, the request will have a *state* value associated with a different session (the attacker’s), so the client can detect the attack and abort the request. |
| Replay of authorized requests to the protected resource | Protected resource | An attacker intercepts an authorized request to the resource server, and then resubmits it. Because the request has a valid authorization header, it may be honored by the protected resource. | * TLS encryption * Require clients to authenticate to the protected resource with a replay-resistant mechanism (i.e., not a static client secret) * Require signed requests with nonces |
| Brute-force guessing of access tokens | Protected resource | An attacker submits repeated requests with different token values in an attempt to guess a valid token | * Use token values with high entropy * Use digitally signed tokens * Implement progressive delays in responding to clients that submit successive invalid access tokens, though this may not be acceptable for clients shared by many users (e.g., web applications) |

## OpenID Connect Threat Model and Known Attacks

Because OpenID Connect is built on OAuth, the attacks and considerations discussed in Section 4.1.1 apply to OpenID Connect as well. The standard OAuth players are all present but take on different roles in the OpenID Connect context, with the end-user, Relying Party, OpenID Provider, and identity API acting as the resource owner, client, authorization server, and protected resource respectively.

OAuth is an authorization protocol, but a number of implementations have attempted to use it to authenticate users to client applications [18]. Authentication was not OAuth’s intended purpose, and using it this way has resulted in a number of security issues, including the ability for relying parties to impersonate users to other relying parties. Access tokens are not meant to securely convey user identity or authentication context information to clients. OpenID Connect includes the ID Token for this purpose. ID Tokens are signed (and optionally encrypted) JWTs, which prevents attackers from using modified or manufactured ID Tokens to authenticate. They are also bound to the OAuth client ID of the RP using the *aud* (audience) parameter, preventing malicious RPs from using issued ID Tokens to impersonate the user at other RPs. In addition to the ID Token, OpenID Connect introduces the UserInfo Endpoint, an OP-hosted protected resource from which RPs can request additional claims about the end user using Access Tokens for authorization.

OpenID Connect supports the Authorization Code and Implicit grant types and flows. In both cases, the ID Token is returned from the OpenID Provider along with the access token. OpenID Connect also adds the hybrid flows, in which the authorization endpoint returns an authorization code along with an id token, an access token for the UserInfo Endpoint, or both. As with the OAuth implicit flow, the hybrid flows expose tokens to the end user’s browser. They are intended to provide flexibility and efficiency for RP implementations. For example, RPs that rarely use the UserInfo Endpoint can submit a single request to the authorization server to obtain the ID token, and an access token in case it later needs to contact the UserInfo Endpoint. The hybrid flows should not be used without consideration of the risks of exposing ID tokens or access tokens to the browser.

### End User Security Issues

End users gain the convenience of single sign-on from the use of OpenID Connect and other identity federation schemes, which eliminates the need to maintain separate login credentials with each RP. This may come at the expense of privacy: RPs may obtain sensitive information about the user from the OP, and the OP could gain insight into the user’s activities on other websites and applications, or at least be able to identify other sites used by the OP. These privacy issues affect most identity federation systems. Scopes can be used to restrict the RP’s authorizations to a subset of user profile information, and OpenID Connect only releases scopes of identity information to the RP that the end user has explicitly authorized.

### Relying Party Security Issues

Using OpenID Connect to authenticate users frees the RP from maintaining and managing account credentials for end users. Relying on an external party to identify and authenticate users requires a significant amount of trust in the OP. RPs should evaluate the OP’s policies, such as the authentication mechanisms supported or required for user authentication, and its overall security posture to determine whether it meets the RP’s requirements for confidence in the asserted identities and claims. In the VA’s case, an agreement to accept federated login from an OP would likely involve a risk assessment and signing of an MOU with the provider.

The risks to an RP in a federation scenario include users subverting the authentication process by submitting intercepted, manufactured, or modified identity tokens or claims in order to gain unauthorized access. OpenID Connect’s use of signed ID tokens and the optional audience parameter mitigate these risks. However, UserInfo Endpoint responses are not required to be signed, so claims obtained from that endpoint may still be subject to manipulation when signatures are not used.

### OpenID Provider Security Issues

An OpenID Provider is an OAuth Authorization server with an additional required endpoint, the UserInfo Endpoint, from which authorized RPs can query claims about users. The OP also may convey information about the user’s authentication context to the RP, such as the specific authentications methods used (e.g., password, one-time password generator, software Public Key Infrastructure (PKI) certificate) or the assurance level.

Theft of ID tokens or UserInfo responses by malicious or compromised RPs is a risk that could enable an attacker to impersonate an authorized end user at other RP applications. The use of optional audience restrictions and client-supplied nonces mitigate the risk of attackers re-using intercepted ID tokens to authenticate to other RPs.

### UserInfo Endpoint Issues

The UserInfo Endpoint is a conventional OAuth 2.0 protected resource, with the same security considerations. Although an ID Token must be signed using JWS, signatures are not required on UserInfo responses, which could enable attackers to modify or manufacture claims in the response.

### Attacks against OpenID Connect

Table 3 below summarizes some known attacks on OpenID Connect and countermeasures that can defend against them. These are described in the OpenID Connect Core specification [13]. The “Against” column indicates the entity or component that is directly targeted by the attack. This table does not include attacks that are common to OpenID Connect and OAuth, covered in Section 4.1.1.6, except for those where OpenID Connect offers additional countermeasures.

Table - Attacks against OpenID Connect

| Attack | Against | Details | Countermeasures |
| --- | --- | --- | --- |
| Authentication request disclosure | Resource owner | Attacker intercepts an authorization request between the RP and OP (e.g., through network address spoofing), obtaining information about the end user’s use of RPs | * TLS Encryption * Require clients to submit requests as encrypted JWTs using the *request* or *request\_uri* parameter |
| Token manufacture or modification | RP | Attacker generates an ID token or modifies an existing one to obtain unauthorized access | * OPs must sign tokens, and RPs must validate signatures |
| ID token or UserInfo response disclosure | OP | Attacker intercepts an OP response containing user claims or sensitive authentication information (e.g., through network address spoofing) | * Use the authorization code flow, which requires clients to authenticate before ID tokens are issued * Encrypt responses using the client’s public encryption key |
| Token Substitution | RP | Attacker substitutes an authorization code or access token with one obtained from a legitimate (or higher-privileged) user in order to authenticate as that user | * ID tokens must include the c\_hash parameter to enable validation of the code returned from the authorization endpoint. If an access token is also returned, the ID token will contain an at\_hash parameter to validate the access token. * Use client-generated nonce values to detect substitution |

## Rationale for Profiling Decisions

The key products of the Secure RESTful Interface Profile task are the standards profiles that were created for OAuth 2.0 [3] and OpenID Connect 1.0 [4]. Profiles define constraints and requirements for implementations of standards to meet functionality, security, or interoperability requirements. The two stated goals for creation of the VA profiles are:

1. Define a mandatory baseline set of security controls suitable for most VA use cases while maintaining reasonable ease of implementation and functionality
2. Identify optional advanced security controls for sensitive use cases where heightened risks justify more stringent controls that would increase the required implementation effort and may reduce or restrict functionality

The OAuth and OpenID Connect profiles provide concise, unambiguous guidance to developers and engineers implementing systems based on these standards. As such, they do not provide much background or discuss the rationale for the decisions made in creating the profiles.

The profiles also require that implementers review the security guidance in the core OAuth specification as well as the Threat Model and Security Considerations document, and implement all applicable recommendations.

Existing OAuth and OpenID Connect profiles developed for two prior initiatives provided a starting point for the profiles developed under this task. Those initiatives are the RESTful Health Exchange (RHEx) [19] and BlueButton+ (BB+) REST [20] pilots – that demonstrated the use of REST and open standards to support secure health information exchange.

The following sections of the document explain the rationale for the design choices that were made in creating the profiles. Throughout section 4.2 and its subsections, references to “the profiles” are meant to indicate the OAuth and OpenID Connect profiles written for the Secure RESTful Interfaces task.

## OAuth Profile

The OAuth 2.0 standards are particularly loose, incorporating several different protocol flows and leaving key details like the content of access tokens up to the implementation. Some controls that help protect against common attacks, such as requiring clients to register their redirection URLs, are optional in the standard. The OAuth profile [3] strikes a balance in addressing these issues without making implementation prohibitively difficult.

### Client Types

The profile is limited to three client types:

* Full client with user delegation – a client capable of securely maintaining client credentials and interacting with the resource owner’s browser, which acts on behalf of a particular resource owner
* Browser-embedded client with user delegation – a client that runs in the resource owner’s browser (e.g., a JavaScript application with no back-end component), and acts on behalf of a particular resource owner
* Direct access client – a client that directly accesses protected resources and does not act on behalf of a particular resource owner.

These client types use the OAuth 2.0 authorization code, implicit, and client credentials authorization grants, respectively. The OAuth specification defines another grant type called the resource owner credentials grant, but this grant is prohibited in the profile because it requires the resource owner to provide his/her credentials directly to the client, which the client exchanges for an access token. It is intended only for transitioning legacy apps into an OAuth authorization scheme.

### Client Authentication

Confidential clients (which includes full clients and direct access clients as defined above) are required to authenticate to the token endpoint using a signed JWT, using the private\_key\_jwt method as specified in OpenID Connect Core [13]. The profile requires the JWT to contain the following claims:

* Issuer(*iss*): the client ID of the client creating the token
* Subject (*sub*): the client ID of the client creating the token
* Audience (*aud*): the URL of the authorization server’s token endpoint
* Issued at (*iat*): the time that the token was created by the client
* JWT ID (*jti*): a unique identifier generated by the client for this authentication. This identifier MUST contain at least 128 bits of entropy and MUST NOT be re-used by any subsequent authentication token.

Below is a sample claim set with the required claims for client authentication under the OAuth profile. This claim set would be used as the JWS payload:

{

"iss":"client1234@example.com",

"sub":"client1234@example.com",

"aud":"https://jwt-rp.example.net",

"iat":1406538000,

"jti":"qNH1IYYSz5darfrysUr/oQ=="

}

The JWT ID claim provides protection against attackers replaying token requests from authorized clients. Note that the authorization server must retain submitted JWT IDs for at least the validity period of the token and check for duplicate values in order for the JWT ID to provide effective replay protection. The *issued at* claim also enables maximum age policies to limit the usefulness of captured requests.

Using signed JWTs provides a form of cryptographic authentication of the client, though it may not be authentication in the strict sense of identifying the client as a known entity in which trust has previously been established. This depends on the controls over client registration. If client registration and credential issuance are strictly controlled and a registry of trusted client keys is maintained, then the JWT signature can provide assurance that the client making the request is trusted. This model does not scale well for native clients, however, where each installed instance has its own credentials. In large-scale implementations, clients typically register their public keys during client registration, though nothing may be known about the client at that time. However, the OAuth client delegation flows link the resource owner’s sessions at the client and the authorization server, providing some assurance that the client making the request to the token endpoint is the same client with which the resource owner is interacting. Future requests signed with the same key pair can be assumed to come from the same client instance. Because the authentication is based on a signature, there is no need to expose a long-term client secret subject to capture on the network. Impersonating the client to the token endpoint requires obtaining the client’s private key.

Using signed JWTs for authentication requires clients to generate their own key pairs, but it does not require the issuance of trusted certificates to every client. Requiring certificates could be a burden for some clients. It could be particularly difficult for native clients, where each installed instance must have its own credentials. Clients register their public keys with the authorization server during client registration, either by providing the keys directly in the *jwks* field, or by providing a *jwks\_uri* where the authorization server can access the client’s key set. Using the *jwks\_uri* method is preferred, since it makes key rotation easier. Registration of keys is necessary to enable the authorization server to validate signatures, and optionally to encrypt data sent to the client. If both signature and encryption are used, the client should generate separate keys for these activities.

### Client Redirect URI Restrictions

The redirect URI is critical to the security of OAuth implementations, since it controls the destination to which the authorization server will send authorization codes; in the implicit flow, the authorization server sends access tokens to the redirect URI. A number of attacks on OAuth involve manipulation of the redirect URI to send codes and tokens to an attacker-controlled destination. To prevent these attacks, the OAuth profile requires clients to register their full redirect URIs with the authorization server, and requires the authorization server to validate all submitted redirect\_uri values against the registered URIs. The full path of the URI must be registered, and URIs must be in one of the following categories:

* Hosted on a website with TLS protection (a HTTPS URI)
* Hosted on the local domain of the client (e.g., http://localhost/)
* Hosted on a client-specific non-remote-protocol URI scheme (e.g., myapp://)

HTTPS is the only approved protocol for redirect URIs not hosted on the local client machine, and clients are required to validate the remote server’s TLS certificate. This reduces the risk of an attacker spoofing the address of the client’s redirect URI, and also takes advantage of TLS encryption and integrity protection.

The profile also prohibits clients from having open redirects on their redirect URIs. An open redirect, considered a vulnerability in and of itself, is an application that accepts a URL as a parameter and redirects the user to that URL without any restrictions or validation. Open redirects in OAuth clients can lead to the exposure of authorization codes (or tokens in the implicit flow) because attackers can specify parameters to redirect responses to servers they control. Redirect URI registration reduces this risk, unless the client makes an open redirect available at one of its registered redirect URIs, since the authorization server only validates the redirect URI path and not the query string. The client developer is responsible for preventing open redirects. The organization hosting the authorization server could check registered redirect URIs for open redirects, but the redirect functionality could be undocumented, hidden, or added at a later time.

### Use of State Parameter

The OAuth profile requires clients to use an unpredictable value for the *state* parameter to prevent token and code substitution attacks. The *state* parameter is submitted by the client in the authorization request. Authorization servers are required to return the same *state* value in the authorization response sent to the client’s redirect URI. Clients must validate that the same value they originally submitted is returned from the authorization server. This prevents attacks such as the authorization code fixation attack listed in Table 2.

Because the OAuth authorization request and response occur outside the client application, the client needs some way of associating an authorization response with the client session in which it was initiated. Substitution attacks are one reason for this; in addition, web application clients may submit several concurrent authorization requests for different users and protected resources. Client session information cannot be conveyed in the URI path, since this would preclude strict validation of the entire redirect URI path against the client’s registered URIs. Instead, the OAuth specification created the *state* query parameter for this purpose.

Clients must use a state value with at least 128 bits of entropy, associate the state with a particular client session, and ensure that the *state* value returned in the authorization response matches the value submitted with the authorization request. This ensures that the returned code or token is associated with the correct client session, and no substitution has occurred.

### Access Token Format

RFC 6749 does not specify a format for the access token. The OAuth profile requires authorization servers to issue access tokens in the form of signed and optionally encrypted JWTs with at least the following required claims:

* Issuer (*iss*): the issuer URL of the server that issued the token
* Audience (*aud*): audience list, an array containing the identifier of the protected resource this token is valid for (MAY be multiple values)
* Authorized party (*azp*):The client id of the client to whom this token was issued
* Subject (*sub*):The identifier of the end-user that authorized this client, or the client id of a client acting on its own behalf (such as a direct access client)
* Key ID (*kid*): the key ID of the keypair used to sign this token
* Expiration (*exp*): expiration time after which the token must not be accepted
* JWT ID (*jti*): A unique JWT ID value with at least 128 bits of entropy. This value MUST not be re-used in another token. Clients MUST check for reuse of *jti* values and reject all tokens issued with duplicate *jti* values.

A sample access token claim set might look like the following; additional claims may be included in the claim set:

{

"iss":"https://jwt-idp.example.com",

"aud":"https://jwt-rp.example.net",

"azp":"client1234@example.com”,

"sub":"jane.doe@example.com",

"kid":"1e9gdk7",

"exp":1406538000,

"jti":"hTNObzOO0Dv/SBLL3veKng=="

}

To access protected resources, clients include these tokens in the *authorization* HTTP header with the “Bearer” authentication scheme, as in the following sample request:

GET /resource HTTP/1.1

Host: server.example.com

Authorization: Bearer mF\_9.B5f-4.1JqM

The OAuth specification requires protected resources to validate access tokens and ensure that their scope includes the requested access, but it does not specify how protected resources should validate tokens, or how they can determine the scope of authorization. Signed JWT access tokens can be validated by validating the signature using the authorization server’s public key and checking the other fields for their expected values. Using signed tokens prevents attacks involving the manufacture or modification of tokens, since the attacker cannot create a valid signature without possessing the authorization server’s private signing key. Clients can detect substitution attacks, because the authorized presenter claim will contain a different client ID than the authenticated client submitting the token. Protected resources can detect attempts to use tokens issued for different resources by checking the audience claim. Replay attacks can be detected by tracking submitted token IDs to detect duplicates.

The use of signed JWTs as access tokens, combined with the restrictions on redirect URIs discussed above, severely reduce the risk of many of the known attacks on OAuth.

### Additional Authorization Server Requirements

The OAuth profile also requires authorization servers to protect communications with all of its OAuth endpoints using TLS with server authentication, to protect tokens, codes, and credentials in transit over the network. The profile also requires authorization servers to provide the following OAuth endpoints, in addition to the authorization and token endpoints:

* Token introspection endpoint – enables protected resources to query the authorization server for metadata about an access token presented by a client (e.g., to find out the authorized scopes associated with the token)
* Token revocation endpoint – enables protected resources to signal to the authorization server that a token is no longer needed (e.g., when the user uninstalls a native client application that has active tokens)
* Discovery endpoint – an endpoint hosted at a well-known URL that returns a JSON object containing the URLs for the authorization server’s endpoints

Providing a registration endpoint to support OAuth dynamic client registration as specified in the draft dynamic registration specification [21] is not required.

### Advanced Security Options

The OAuth profile strikes a balance between security on the one hand, and usability and ease of implementation on the other. In some cases, heightened security concerns may justify more aggressive security measures. The OAuth profile mentions some advanced security options that may not be practical in all cases, but which can provide additional protection for use cases where it is needed.

Requiring OAuth clients to authenticate to protected resources is one such option. In many commercial OAuth implementations, presentation of an access token is sufficient to gain access to a protected resource. Requiring clients to authenticate, combined with the JWT access token specified in the profile, would enable protected resources to validate that the client presenting the token is the one to which it was issued by the authorization server, by comparing the authenticated client ID with the authorized party claim in the access token. This would limit the usefulness of a stolen access token in many cases, since the token could only be presented by the client to which it was issued. Implementing this control requires the protected resource to implement token validation.

Another option is to require clients to authenticate to the token endpoint and to protected resources using mutual TLS authentication using PKI certificates. PKI authentication provides strong protection against Man-in-the-Middle (MITM) attacks, since the attacker cannot successfully complete the TLS handshake with the server when brokering the connection for another client. TLS client certificate authentication is also natively supported by many common web servers, whereas JWT validation would typically be handled in application code. In practice, PKI has proven difficult to implement at large scale, particularly in the scope of information exchange with external parties on which the VA Secure RESTful Interface Profile is focused. However, in cases where the clients already possess or can easily be provisioned with PKI credentials, an OAuth implementation can take advantage of them for strong client authentication.

The OAuth profile also mentions a set of draft standards for OAuth proof of possession tokens [17]. Most existing OAuth implementations use bearer tokens. Bearer tokens could be compared to physical tickets purchased at a movie theater. Customers exchange money at the ticket counter for physical tickets, which entitle the person holding them to watch a movie. At the entrance to the theater, an usher will admit anyone who arrives with a valid movie ticket. It is the customer’s responsibility to maintain control of the ticket and safely deliver it to the usher. The usher cannot demand that customers prove that they were the original purchasers of the tickets in their hands, nor could the customers easily prove it.[[6]](#footnote-6)

Proof of possession tokens provide a mechanism for the presenter of a token to prove their rightful ownership of it by demonstrating possession of a cryptographic key. The client must authenticate cryptographically to the authorization server’s token endpoint; both symmetric and asymmetric algorithms are supported. This proves to the authorization sever that the client possesses a particular key. When generating the access token, the authorization server includes a confirmation (*cnf*) claim with information about the cryptographic key possessed by the client. When a protected resource receives the access token, it can challenge the client to demonstrate possession of the key (e.g., by signing or creating a Hashed Message Authentication Code, or HMAC). This provides a stronger binding of the claims in the access token to the client presenting it. An attacker could not use a stolen proof of possession token to access a resource without also obtaining the key from the legitimate client.

## OpenID Connect Profile

Because OpenID Connect is based on OAuth, the requirements of the OAuth profile apply to OpenID Connect as well, including the use of signed JWT access tokens. In general, the OpenID Connect 1.0 standard leaves fewer options up to implementers and makes more security features mandatory than does OAuth. For both of these reasons, the OpenID Connect profile is quite a bit shorter than the OAuth profile.

### ID Token Requirements

The OpenID Connect Core standard requires OpenID Providers to sign ID Tokens, but permits relying parties to omit signature validation when the ID Token is received over a direct HTTPS connection from the client to the OpenID Provider. The OpenID Connect profile requires relying parties to always validate signatures on ID tokens, and requires ID tokens to expire within five minutes of issuance. The profile also requires relying parties to verify the following claims in the ID Token:

* *issuer:* The “iss” field is the URL of the expected issuer
* *audience:* The “aud” field contains the client ID of the relying party
* *expiration, issued-at, not-before:* The “exp”, “iat”, “nbf” fields are dates within acceptable ranges

Validation of these claims, along with token signature validation, prevents attacks involving the manufacture or modification of ID tokens, and the use of ID tokens issued for one relying party at a different relying party. The expiration and other date-time claims limit the period during which an intercepted ID token could be used.

### UserInfo Requirements

The OpenID Connect profile requires OpenID Providers to provide UserInfo endpoints. Relying parties can access the UserInfo endpoint using an access token returned from the OpenID Provider’s token endpoint in order to request additional claims about the user or the authentication context, which includes information about how the user authenticated to the OpenID Provider, beyond the claims that were provided in the initial ID Token. Relying parties can request specific claims by including them as *scope* parameters in the authorization request. As with other uses of OAuth scopes, the resource owner (the user being authenticated in this case) may approve a subset of the requested scope, in which case not all requested claims will be returned. This maintains the user’s control over which of their user profile attributes are shared with the relying party.

While OpenID Connect requires ID Tokens to be signed JWTs, it does not require UserInfo responses to be signed. The standard does require all communications between the relying party and UserInfo endpoint to use TLS, which provides encryption and integrity protection of transmitted data. The OpenID Connect profile requires OpenID providers to support signing and encryption of UserInfo responses using the JOSE standards, for use cases where message-level protection is required (e.g., to protect the integrity of user claims through a connection where TLS is terminated before the final destination of the UserInfo response). Signatures protect the integrity and authenticity of user claims, preventing attackers from creating claims or modifying them in transit. Encrypting user claims to prevent unauthorized disclosure may be required when they contain sensitive data about the user.

### Authorization Request Objects

OpenID Connect permits relying parties to submit authentication request parameters as claims in an optionally signed or encrypted JWT. In the traditional OAuth method, they are submitted as individual parameters in the URL query string, but OpenID Connect request objects are submitted as a single parameter containing the JWT or as a URI pointing to the JWT containing the request. Request objects enable the relying party to sign or encrypt the authentication request parameters. As with JOSE-protected UserInfo responses, these messages are already protected by a TLS connection between the client and authorization endpoint, but message-level signatures and encryption may be required, particularly in cases where the TLS connection is terminated by an intermediary device between the message sender and the destination.

### Authentication Context Claims

In many use cases, VA relying parties may need information about the user’s authentication context – for example, the type of credential(s) used to authenticate – in order to make an access control decision. The OpenID Connect specification defines two standard claims that enable the OP to communicate this information to the RP: “acr” (authentication context class reference), and “amr” (authentication methods reference).

When sharing this type of claim across organizational boundaries, it is important for federation partners to agree to terms and a common scheme for how authentication context will be conveyed. The concept of authentication context class is roughly analogous to the National Institute of Standards and Technology’s (NIST’s) notion of Level of Assurance (LOA), so in the interest of re-using an established standard, the OpenID Connect profile recommends using the established URIs for NIST LOAs published by FICAM. Unfortunately, there is not a comparable existing standard for the authentication methods reference claim, which indicates the set of actual authentication mechanisms used to authenticate the user (e.g., a multi-factor login scheme might use password and a one-time code sent to the user’s mobile device). It is recommended that VA should work with its information sharing partners to define a common set of values for the authentication methods reference field.

## Security Policy Analysis

This section presents an analysis of applicable VA and Federal IT policies relevant to the Secure RESTful Interface Profile work, and identifies specific areas where policy may impact the implementation of a REST security infrastructure. One of the goals of this analysis was to identify any substantive conflicts between the recommendations made in the profiles and existing policy. No such conflicts were identified. The profiles deal mainly with specifics of the OAuth and OpenID Connect standards at a level not typically addressed by policy documents. When it comes to the actual implementation and operation of the REST security infrastructure, however, a number of policies will apply. This document identifies some policies that may significantly impact a VA implementation of OAuth and OpenID Connect.

## Relevant Policy Documents

The task team reviewed number of VA policy documents, NIST Special Publications (SP), Federal Information Processing Standards (FIPS), and other documents for content applicable to the Secure RESTful Interfaces task. Table 4 lists documents with applicable content.

Table - Relevant Policy Documents

| Reference | Title | Applicability |
| --- | --- | --- |
| VA Handbook 6500 [22] | RISK MANAGEMENT FRAMEWORK FOR VA INFORMATION SYSTEMS – TIER 3: VA INFORMATION SECURITY PROGRAM | Core VA policy document |
| Office of Management and Budget (OMB) Memo 04-04 [23] | E-Authentication Guidance for Federal Agencies | Guidance on assessing the required assurance level for online transactions |
| NIST SP 800-63-2 [24] | Electronic Authentication Guideline | Guidance on authentication mechanisms and protocols to meet assurance requirements per OMB M-04-04 risk assessment |
| NIST SP 800-53 [25] | Recommended Security Controls for Federal Information Systems and Organizations | Required security controls based on risk categorization for Government systems |
| NIST SP 800-47 [26] | Security Guide for Interconnecting Information Technology Systems | Standards for establishing, managing, and securing system interconnections |

## Policy Considerations

### OAuth Clients and System Interconnections

NIST SP 800-47 provides guidance for establishing and managing system interconnections, and requires that the interconnecting parties sign a Memorandum of Understanding (MOU), come to an agreement on how the security of the connection will be managed by each side, and sign an Interconnection Security Agreement (ISA). SP 800-47 defines a system interconnection somewhat broadly as “the direct connection of two or more IT systems for the purpose of sharing data and other information resources.” The description of control CA-3 (System Interconnections) in SP 800-53 provides additional guidance:

This control applies to dedicated connections between information systems (i.e., system interconnections) and does not apply to transitory, user-controlled connections such as email and website browsing.

It is not entirely clear which types of OAuth clients, if any, would be considered interconnected systems with the authorization server or protected resources. A user accessing a Government system with a web browser through a public interface is not considered a system interconnection, and certainly meets the “transitory, user-controlled” condition stated in SP 800-53. OAuth clients using the implicit flow to access resources over a public interface are basically equivalent to the browser user.

Native clients deviate from the 800-53 language, since they may access resources when the user is not present. The same is true of web clients, and it seems more reasonable to consider a large hosted web application to be an “interconnected system.” It would be impractical for the Government to sign MOUs and ISAs with every person who installs a copy of a native application. Attempting to treat web application clients as interconnected systems would be more manageable, since a large number of users might access the system through a single client.

However, the three OAuth client types listed above are all acting on behalf of individual users, and they all access resources through public interfaces. In this sense, they are substantially similar to a single user accessing resources with a web browser. The main difference is that they may submit automated requests when the user is not present. Web browser users might also decide to script their requests to run unattended using a tool such as wget[[7]](#footnote-7), but does this act create a system interconnection?

OAuth clients using the client credentials grant, on the other hand, do not obtain authorization from particular users or act on their behalf, and are often used to perform bulk transfers between systems. Though they may use the same public interfaces as the other OAuth client types, they would seem to fit the definition of interconnected systems and require the management processes defined in NIST SP 800-47.

The 800-53 language exempting “user-controlled, transitory” connections from the definition of system interconnections supports the argument that native and browser-based OAuth clients, and perhaps even web application clients, should not be considered interconnected systems. Partner systems performing bulk transfers using the client credentials grant, on the other hand, do not act on behalf of particular users and fit the definition of interconnected systems. Updated policy guidance at the Federal Government level would help clarify these issues. The MITRE task team recommends that VA raise these issues with the FICAM program office and other cross-government policy organizations (see Section 6.2).

### Level of Assurance Requirements for Access to Patient Data

OMB M-04-04 includes a framework for performing a risk assessment to determine the required LOA for authentication of a given transaction. The LOA determination is based on the potential impact of unauthorized access. However, the framework requires a subjective evaluation of distress, harm, and loss caused by authentication failures using adjectives such as “limited,” “serious,” and “severe.” Perhaps unsurprisingly, there is no consensus on LOA requirements for providers or patients to access electronic health record information. The MITRE task team recommends that VA work with the Federal Health Architecture, the Office of the National Coordinator, and other mission partners to arrive at a common set of assurance requirements for use cases involving patient data.

### Public-key Cryptography without X.509 Certificates

OpenID Connect uses the JWT standard for ID Tokens, and the Secure RESTful Interfaces profile for OAuth specifies the use of JWT for client authentication and access tokens. JWTs use the JOSE set of standards for signatures and encryption, which refer to JWK to specify how cryptographic keys are referenced and shared among participants.

JWK supports the use of X.509 certificates to bind keys to systems and other entities, but does not require it; keys can also be passed by value in JSON objects, or by reference through URLs indicating where the JSON objects containing keys can be downloaded.

A great deal of federal IT policy is based on the proper use of PKI and X.509, detailing how trust chains should be built, certificates should be validated, and revocation information should be obtained. For some OAuth use cases, the use of X.509 certificates is impractical. Native clients, for example, must use unique credentials for each installed instance of the client software, and it would be impractical to issue a certificate signed by an established CA to each one. More to the point, such a certificate would be of little value, since nothing would be known about its recipient except that he/she had installed a particular software package. The public/private key pair, however, could still be used to provide assurance that a client presenting the same key in future sessions is the same client that initially registered the public key with the authorization server.

Federal implementation guidance on JOSE and related standards would help to ensure that implementations by Government agencies are secure and interoperable.

# Ongoing Task Team Work

The completion of the draft standards profiles and the security analysis concludes Phase 1 of the Secure RESTful Interface Profile task. Phase 2 of this task is intended to include a pilot implementation to validate the usability and security of the profiles and continuing outreach to stakeholders within VA and in partner organizations.

## Pilot Implementation

The pilot implementation is currently in the planning stages. The task team has defined use cases to demonstrate multiple variations of the RESTful security patterns in this document, and infrastructure requirements to support the use cases, including VA and partner instances of OAuth and OpenID Connect service providers and EHR systems. The team will soon begin provisioning virtual machines and installing the required software.

## Ongoing Outreach Efforts

The task team has maintained ongoing communication with stakeholders in VA, the Office of the National Coordinator for Health IT (ONC), and the wider IT standards community, and is discussing some potential opportunities for collaboration.

VHA is currently running multiple pilots dealing with patient privacy and consent, data provenance, data tagging, and Fast Health Interoperability Resources (FHIR) interfaces, with plans to include OAuth and UMA. Also, ONC is working with the Massachusetts Institute of Technology (MIT) Kerberos Consortium with OAuth and OpenID Connect, working toward a standard profile for health information exchange to present at next year’s Healthcare Information and Management Systems Society (HIMSS) conference. Both efforts are closely related to the Secure RESTful Interface Profile work. The team will continue to coordinate with these other initiatives and seek opportunities to pool resources and collaborate.

# Summary and Guidance on Next Steps

The RESTful architecture style is being aggressively pursued through multiple initiatives within the VA and by its partners in ONC and standards bodies such as Health Level 7 (HL7). In the course of this analysis, MITRE identified multiple initiatives to provide a RESTful interface to the Veterans Health Information Systems and Technology Architecture (VistA) within the VA, with others underway in the open source VistA community. The VA is in need of a framework for securing RESTful interfaces soon, before a large number of services are deployed that will later need to be re-engineered to meet security requirements.

The draft profiles developed under the Secure RESTful Interfaces task provide a solid baseline level of security by constraining OAuth and OpenID Connect to address a number of security concerns. The profiles are a starting point for developing a VA RESTful security framework. This section summarizes MITRE task team recommendations on subsequent steps that VA could take towards adoption of the secure RESTful interface profiles.

## Steps towards Adoption of the Standards Profiles

The following are recommended steps for VA to take towards adoption and implementation of the open standards profiles and security guidance produced under the Secure RESTful Interfaces task.

* **Host Additional Discussions with VA Stakeholders** – Brief additional VA stakeholders including Product Development, Cybersecurity, the Enterprise Shared Services (ESS) Security Working Group, and the Identity and Access Management (IAM) support team, on the profiles and work towards general consensus on an approach to security RESTful interfaces.
* **Sponsor Discussions with External Partners** – Work with partners in the DoD, the Federal Health Architecture (FHA), and the health IT community towards a common standard for REST security, using the Secure RESTful Interface profiles as a starting point.
* **Perform Technical Integration Analysis** – Determine how to integrate enterprise OAuth authorization services and OpenID Connect Providers into VA’s planned IAM and Service-Oriented Architecture (SOA) infrastructure. Some considerations include where to implement OAuth and OpenID Connect capabilities (e.g., integrated into the existing external single sign-on (SSOe) infrastructure or built separately), and how to convey user identity attributes to back-end systems through the Enterprise Service Bus (ESB).
* **Support Client Developers** – Plan to support developers of OAuth-enabled client software wishing to integrate with OAuth and OpenID Connect services with documentation, non-Production providers with which to test, and a mechanism for client registration. Google’s developer guide for using their OAuth provider [27] is a good example of the documentation developers will need.
* **Make Client Standards and Registration Process Transparent –** Define standards and acceptance criteria for clients developed within VA and externally. Define client vetting (if applicable), registration, support, and credential management processes, and explain the requirements and process in developer documentation.
* **Incorporate REST Security Standards Profiles and Other Guidance into Enterprise Architecture (EA)** – Publish guidance in EA web portals. Update the VA Enterprise Technical Architecture (ETA) compliance criteria with requirements for REST security.

## Summary of Identified Issues and Recommendations

The MITRE task team identified the following issues in the policy analysis and the definition of the draft OpenID Connect profile. This section summarizes the issues and presents recommendations to address them.

* **Need for Common Authentication Methods Reference Identifiers among VA’s Sharing Partners –** As discussed in Section 4.2.2.4, an OpenID provider can convey information about the End User’s authentication mechanism – for example, a password, a one-time password generator, or a smart card - to the Relying Party using the *amr* claim. Currently, there is no standard registry of values for these claims. MITRE recommends that the VA work with its information sharing partners to define a common scheme for conveying authentication token information in the Authentication Methods Reference (*amr*) claim.
* **Need for Common Understanding of LOA Requirements for Use Cases involving Medical Data** – currently, there is not wide agreement on the NIST level of assurance (LOA) required for typical use cases involving medical data (e.g., a patient accessing his/her own records, a specialist accessing a referred patient’s records). This poses the risk that authentication requirements will be inconsistently applied and data may not be appropriately protected. MITRE recommends that VA work with the Federal Health Architecture, the Office of the National Coordinator, and other mission partners to arrive at a common set of assurance requirements for use cases involving patient data.
* **Need for Federal Policy Guidance** – Federal security policy as defined in NIST SP 800-47 and SP 800-53 defines requirements for system interconnections, including the signing of MOUs and ISAs between the organizations hosting interconnected systems. The variety of types of OAuth clients with varying capabilities makes it unclear which classes of clients would be considered interconnected systems. Also, much of the Federal IT policy on cryptography focuses on the use of X.509 certificates, but the emerging JOSE set of standards can employ public key cryptography without the use of certificates. Policy guidance on the use of JOSE for encryption and digital signatures is needed to ensure that Government implementations are both secure and interoperable. MITRE recommends that VA engage with NIST and other Federal agencies to develop consensus on these issues and request clarifying updates to policy guidance.

# List of Acronyms

| Acronym | Definition |
| --- | --- |
| AE | Authorization Endpoint |
| API | Application Programming Interface |
| ARP | Address Resolution Protocol |
| ASD | Architecture, Strategy, and Design |
| BB+ | BlueButton+ |
| CSRF | Cross-Site Request Forgery |
| DNS | Domain Name System |
| DoD | Department of Defense |
| DoS | Denial of Service |
| DS Logon | DoD Self-service Logon |
| EA | Enterprise Architecture |
| EHR | Electronic Health Record |
| ESB | Enterprise Service Bus |
| ESS | Enterprise Shared Services |
| ETA | Enterprise Technical Architecture |
| FHA | Federal Health Architecture |
| FHIR | Fast Healthcare Interoperability Resources |
| FIPS | Federal Information Processing Standard |
| HIMSS | Healthcare Information and Management Systems Society |
| HL7 | Health Level Seven |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | HTTP Secure |
| IAM | Identity and Access Management |
| IETF | Internet Engineering Task Force |
| IP | Internet Protocol |
| ISA | Interconnection Security Agreement |
| IT | Information Technology |
| JOSE | JSON Object Signing and Encryption |
| JSON | JavaScript Object Notation |
| JWA | JSON Web Algorithms |
| JWE | JSON Web Encryption |
| JWK | JSON Web Key |
| JWS | JSON Web Signature |
| JWT | JSON Web Token |
| LOA | Level of Assurance |
| MAC | Message Authentication Code |
| MIT | Massachusetts Institute of Technology |
| MITM | Man-in-the-Middle |
| MOU | Memorandum of Understanding |
| NIST | National Institute of Standards and Technology |
| OIDF | OpenID Foundation |
| OIT | Office of Information & Technology |
| OMB | Office of Management and Budget |
| ONC | Office of the National Coordinator for Health IT |
| OP | OpenID Provider |
| PGD | Patient-Generated Data |
| PKI | Public Key Infrastructure |
| REST | Representational State Transfer |
| RHEx | RESTful Health Exchange |
| RP | Relying Party |
| SAML | Security Assertion Markup Language |
| SOA | Service-Oriented Architecture |
| SP | Special Publication |
| SSOe | Single Sign-on External |
| TAXII | Trusted Automated Exchange of Indicator Information |
| TE | Token Endpoint |
| TLS | Transport-Layer Security |
| UE | UserInfo Endpoint |
| UMA | User-Managed Access |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| VA | Veterans Affairs |
| VistA | Veterans Health Information Systems & Technology Architecture |
| VSO | Veterans Service Organization |

# References

|  |  |
| --- | --- |
| [1] | Department of Veterans Affairs, "FY 2014-2020 Strategic Plan," March 2014. [Online]. Available: http://www.va.gov/op3/docs/StrategicPlanning/VA2014-2020strategicPlan.PDF. [Accessed May 2014]. |
| [2] | The MITRE Corporation, "Secure RESTful Interfaces: Business-oriented Use Cases & Associated Distributed Security Requirements," May 2014. |
| [3] | The MITRE Corporation, "Secure RESTful Interfaces: Draft Profiles for the Use of OAuth 2.0," June 2014. |
| [4] | The MITRE Corporation, "Secure RESTful Interfaces: Draft Profiles for the Use of OpenID Connect," June 2014. |
| [5] | T. Dierks and E. Rescorla, "The Tansport Layer Security (TLS) Protocol Version 1.2," August 2008. [Online]. Available: http://tools.ietf.org/html/rfc5246. [Accessed May 2014]. |
| [6] | D. Hardt, "The OAuth 2.0 Authorization Framework," October 2012. [Online]. Available: http://tools.ietf.org/html/rfc6749. [Accessed May 2014]. |
| [7] | Ecma International, "ECMA-404: The JSON Data Interchange Format," October 2013. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. [Accessed July 2014]. |
| [8] | M. Jones, "JSON Web Signature (JWS)," 4 July 2014. [Online]. Available: http://tools.ietf.org/html/draft-ietf-jose-json-web-signature-31. [Accessed July 2014]. |
| [9] | M. Jones and J. Hildebrand, "JSON Web Encryption (JWE)," 4 July 2014. [Online]. Available: http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-31. [Accessed July 2014]. |
| [10] | M. Jones, "JSON Web Key (JWK)," 4 July 2014. [Online]. Available: http://tools.ietf.org/html/draft-ietf-jose-json-web-key-31. [Accessed July 2014]. |
| [11] | M. Jones, "JSON Web Algorithms (JWA)," 4 July 2014. [Online]. Available: http://tools.ietf.org/html/draft-ietf-jose-json-web-algorithms-31. [Accessed July 2014]. |
| [12] | M. Jones, J. Bradley and N. Sakimura, "JSON Web Token (JWT)," 4 July 2014. [Online]. Available: http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-25. [Accessed July 2014]. |
| [13] | N. e. a. Sakimura, "OpenID Connect Core 1.0," 25 February 2014. [Online]. Available: http://openid.net/specs/openid-connect-core-1\_0.html. [Accessed May 2014]. |
| [14] | T. Hardjono, "User-Managed Access (UMA) Profile of OAuth 2.0," 20 July 2014. [Online]. Available: http://tools.ietf.org/html/draft-hardjono-oauth-umacore-10. [Accessed May 2014]. |
| [15] | The MITRE Corporation, "TAXII Version 1.1," [Online]. Available: http://taxii.mitre.org/specifications/version1.1/. [Accessed May 2014]. |
| [16] | T. Lodderstedt, M. McGloin and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations," January 2013. [Online]. Available: http://tools.ietf.org/html/rfc6819#page-46. [Accessed May 2014]. |
| [17] | P. Hunt, J. Richer, W. Mills, P. Mishra and H. Tschofenig, "OAuth 2.0 Proof-of-Possession (PoP) Security Architecture," 3 April 2014. [Online]. Available: http://tools.ietf.org/html/draft-hunt-oauth-pop-architecture-00#section-1. [Accessed June 2014]. |
| [18] | J. Bradley, "The problem with OAuth for Authentication," 28 January 2012. [Online]. Available: http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html. [Accessed July 2014]. |
| [19] | S&I Framework, "RHEx - Powering Secure, Web Base Health Data Exchange," [Online]. Available: http://wiki.siframework.org/RHEx. [Accessed June 2014]. |
| [20] | S&I Framework, "BlueButton Plus Initiative," [Online]. Available: http://wiki.siframework.org/BlueButton+Plus+Initiative. [Accessed 2014 June]. |
| [21] | J. Richer, M. Jones, J. Bradley, M. Machulak and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol," 22 May 2014. [Online]. Available: http://tools.ietf.org/html/draft-ietf-oauth-dyn-reg-17. [Accessed June 2014]. |
| [22] | Department of Veterans Affairs, *VA HANDBOOK 6500: RISK MANAGEMENT FRAMEWORK FOR VA INFORMATION SYSTEMS – TIER 3: VA INFORMATION SECURITY PROGRAM,* September 2012. |
| [23] | Office of Management and Budget, "M-04-04: E-Authentication Guidance for Federal Agencies," 16 December 2003. [Online]. Available: http://www.whitehouse.gov/sites/default/files/omb/memoranda/fy04/m04-04.pdf. [Accessed July 2014]. |
| [24] | NIST, "SP 800-63-2, Electronic Authentication Guidance," August 2013. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-2.pdf. [Accessed July 2014]. |
| [25] | NIST, "SP 800-53 - Security and Privacy Controls for Federal Information Systems and Organizations, Revision 4," February 2013. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-53r4. [Accessed July 2014]. |
| [26] | NIST, "SP 800-47: Security Guide for Interconnecting Information Systems," August 2002. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-47/sp800-47.pdf. [Accessed July 2014]. |
| [27] | Google Inc., "Using OAuth 2.0 to Access Google APIs," 17 July 2014. [Online]. Available: https://developers.google.com/accounts/docs/OAuth2. [Accessed July 2014]. |
| [28] | Ecma International, "History of Ecma," [Online]. Available: http://www.ecma-international.org/memento/history.htm. [Accessed July 2014]. |

1. ECMA was originally an acronym standing for the European Computer Manufacturers Association, but the organization changed its name in 1994 to Ecma International to reflect its global focus [27]. [↑](#footnote-ref-1)
2. Diagram source: https://kantarainitiative.org/confluence/display/uma/Home [↑](#footnote-ref-2)
3. Domain Name Service (DNS) cache poisoning is an attack in which a network service that translates host names to Internet Protocol (IP) addresses is manipulated to return incorrect results, redirecting requests for legitimate host names to an attacker-controlled machine. Address Resolution Protocol (ARP) cache poisoning is a similar attack against a network service that translates Media Access Control (MAC) addresses to IP addresses. [↑](#footnote-ref-3)
4. Cross-site request forgery, a category of attacks in which a malicious site or application submits unauthorized HTTP requests to another site to which the victim has an active, authenticated session in the same browser, thereby performing actions with the victim’s authorities on the target site [↑](#footnote-ref-4)
5. Note that in order to validate that access tokens are presented by authorized clients, the protected resource must require client authentication. [↑](#footnote-ref-5)
6. Receipts are subject to being lost or stolen along with movie tickets. [↑](#footnote-ref-6)
7. <https://www.gnu.org/software/wget/> [↑](#footnote-ref-7)