# Authentication/Authorization with Cross-Device Flows

## 3.1 Introduction

The rapid proliferation and functional evolution of smartphones continues to change our lives in significant ways. We now use smartphones in every aspect of our daily lives. Authentication and authorization, which are crucial for ensuring that we can use Internet-based services safely, are no exception. In this article, I explain a smartphone-based authentication/authorization method called cross-device flows[*1], something that has been attracting attention in recent years.

A cross-device flow is an authentication/authorization method in which the device (e.g., a PC or smart TV) on which a service is used is separate from the device (e.g., a smartphone) that handles the service authentication/authorization. Say, for instance, that you want to stream video on your smart TV, but that entering your user ID and password into the TV's remote control is awkward, so you use your smartphone instead.

In this case, the cross-device flow solves the problem of using a service on a device with a limited input interface. Cross-device flows are needed in many other situations as well, with a wide range of use cases being proposed. You might, for instance, want to use a service via a device on which you want to avoid entering confidential information, such as a shared or public device. Or you might want to add multi-factor authentication to an existing authentication/authorization flow. Or perhaps you want to perform authentication/authorization on multiple devices using the same private key, but you want to avoid copying that private key.

A number of cross-device flow standards specifications exist, including some that are under development, each with different use cases. Below are some major ones, at which we will take a closer look.

- OAuth 2.0 Device Flow
- OpenID Connect CIBA Flow
- OID4VP's Cross Device Flow
- SIOP v2's Cross-Device Self-Issued OP
- CTAP v2.2's Hybrid transports

## 3.2 OAuth 2.0 Device Flow

OAuth 2.0 Device Authorization Grant (RFC8628)[*2] is an OAuth 2.0 authorization flow. It was standardized by the IETF in 2019. It is commonly called Device Flow. This cross-device flow was designed to allow other devices to be used to assist with applications running on devices with limited user input capabilities, such as smart TVs, digital photo frames, and printers. The case of using a video streaming app on a smart TV mentioned in the previous section is a prime example of this.

Device Flow is an authorization flow. The protocol is designed such that an authorization server issues access tokens that allow client applications to use a service (usually provided as an API). Since it is not an authentication flow, the Device Flow specification does not encompass functionality by which client applications can authenticate end users (functionality for identifying end users, such as the issuance of ID tokens). If you want to perform authentication as well, you need to combine it with something like OpenID Connect.

*1 Cross-Device Flows: Security Best Current Practice (https://datatracker.ietf.org/doc/draft-ietf-oauth-cross-device-security/).
*2 RFC 8628: OAuth 2.0 Device Authorization Grant (https://datatracker.ietf.org/doc/rfc8628/).

Figure 1 is an example of the Device Flow authorization flow. In OAuth 2.0, the application that uses the service is called the client, and the application that performs authorization (usually a web browser) is called the user agent.

1. The end user launches the client on the device.
2. The client sends an authorization request to the authorization server (a).
3. In response, the authorization server returns a device verification code (device code), an end user verification code (user code), and a verification URL for the end user to access.
4. The client displays on screen the user code and verification URL that it received. Verification URLs are usually displayed in the form of QR codes.
5. The end user scans the QR code with a smartphone or the like (b) to obtain the verification URL.
6. The user visits the verification URL via the user agent. The user is asked to authenticate and thus signs in.
7. After signing in, a user code is displayed on screen. (In some cases, the end user is required to enter the user code).
8. While the end user is working with the user agent, the client repeatedly sends access token requests to the authorization server. The requests include the device code as a parameter.
9. The end user confirms that the user code displayed by the client and the user code displayed by the user agent match, confirms any other notes displayed, and then approves (c).
10. The authorization server issues an access token and returns it to the client in response to the access token request (d).

A major difference between Device Flow and other OAuth 2.0 authorization flows is how the front channel is implemented. The term front channel refers to the link between the client and the user agent. Authorization Code Flow as defined in the OAuth 2.0 Authorization Framework (RFC6749)[*3] is the most commonly used OAuth 2.0 authorization flow and works by redirecting the front channel (using HTTP redirects or redirects that use inter-application linking mechanisms such as deep links (Universal Links on iOS and App Links on Android)). But with Device Flow, redirects cannot be used because the client and user agent run on different devices, so instead, the end user acts as an intermediary by scanning a QR code or reading off and manually entering a code.

Creating a Device Flow front channel is simple and does not require specialized hardware, so it is easy to implement, yet it offers less-than-robust security in some respects. It may be susceptible to access token theft via social engineering or man-in-the-middle attacks, and users could be redirected to malicious sites. So it's generally thought that Device Flow should be avoided for clients that access sensitive or important data.

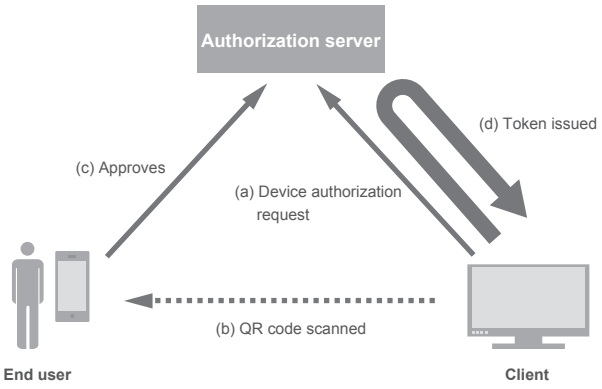


**Figure 1: Example of Device Flow-based Authorization Flow**

*3 RFC 6749 - The OAuth 2.0 Authorization Framework (https://datatracker.ietf.org/doc/rfc6749/).

## 3.3 OpenID Connect CIBA Flow

OpenID Connect Client-Initiated Backchannel Authentication Flow[*4] is an OpenID Connect authentication/authorization flow, abbreviated as CIBA. It was standardized by the OpenID Foundation in 2021. Like Device Flow, CIBA is a cross-device flow that allows the client that uses a service to be on a different device from the one that handles authorization. It is conceptually very different, however. With Device Flow, a single end user operates both devices in most cases, but CIBA was designed with cases in which each device is operated by a different user in mind. This opens up the following sort of use cases for CIBA.

- When a call center rep needs to obtain information from a customer over the phone. In this case, the customer gives their member number to the rep, who then searches for it in a customer management system. A notification is then sent to the customer's smartphone, with a prompt for permission to disclose personal information. The customer management system displays the customer's information to the rep only after the customer provides permission. This mechanism can prevent information breaches caused by staff viewing customer information without permission.
- When approving credit card payments at a store. In this case, when a customer tries to pay via credit card at the cash register, a notification appears on the customer's smartphone with a message confirming the payment details. The payment is completed once the customer approves the message. This offers a more reliable way of identifying people and obtaining consent than asking for a signature or PIN.

Let's take a closer look at CIBA to see how authentication and authorization are implemented (Figure 2). First, some terminology. In CIBA, the device that runs the client is called the consumption device, and the device on which the end user performs authentication is called the authentication device. The authentication device is typically a smartphone. CIBA does not define a term for the application that performs the permissioning operations on the authentication device, but for convenience, I will refer to it as the authentication application. CIBA is an OpenID Connect authentication/authorization flow, so it issues an ID token together with an access token. The server that issues these is called the OpenID Provider (OP).

1. The client sends an authentication request to the OP (a). The request contains a parameter identifying the end user.
2. The OP returns an authentication request ID in response to the authentication request.
3. The OP searches the end user database for an authentication device associated with the end user and then sends a message requesting consent to that authentication device (b). Push notifications (Apple Push Notification Service or Firebase Cloud Messaging) are often used here.
4. The authentication device that receives the consent request starts the authentication application and displays the message on screen.
5. The end user chooses to either consent or decline, and this response is sent to the OP (c).
6. If the end user consents, the OP will issue an access token and an ID token.
7. The client polls the token endpoint and obtains a token (d). The request here includes the authentication request ID as a parameter. If the client is able to expose a notifications endpoint, there is also the option of receiving notifications when a token is issued without any polling.

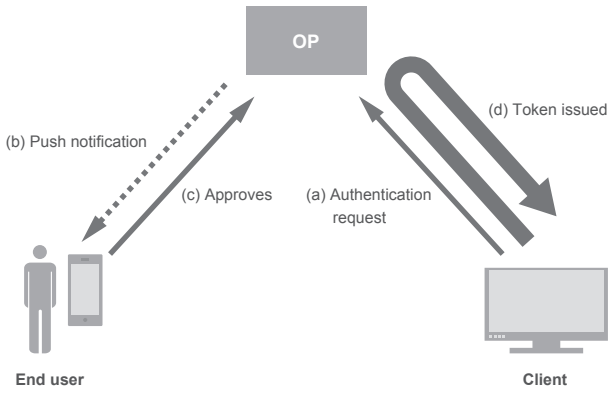


**Figure 2: Example of CIBA Authentication/Authorization Flow**

*4 OpenID Connect Client-Initiated Backchannel Authentication Flow - Core 1.0 (https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html).

The CIBA specification does not define a protocol for communications between the OP and the authentication device. Both the communications method and message specifications are left up to the implementer.

CIBA differs considerably from Device Flow and the other cross-device flows discussed below in that it does not use the front channel; everything is completed via the back channel. The back channel is where interactions between the client and the OP and between the authentication application and the OP occur. Because there is no direct interaction between the client and the authentication application, it supports use cases in which the consumption device and authentication device are separated geographically, as in the call center example above.

Another major feature of CIBA is that it is a client-initiated authentication/authorization flow. With other OAuth/OpenID Connect flows, when a client attempts to access end user resources, authentication/authorization is carried out a single time and the client then holds the token for a lengthy period of time. CIBA makes it possible to issue short-term tokens for each client request, enabling more flexible resource protection.

CIBA is thus quite valuable in that it supports use cases that can be difficult to handle with other authentication/authorization flows. It is attracting attention from the financial industry in particular, and it has also been incorporated into FAPI (an OAuth/OpenID Connect profile for areas that require strong security, such as finance)[*5], which the OpenID Foundation is working to popularize[*6].

## 3.4 OID4VP's Cross Device Flow

This section describes OpenID for Verifiable Presentations[*7] (abbreviated OID4VP), currently being developed by the OpenID Foundation. Before diving into OID4VP, I will briefly explain verifiable credentials, which are used in OID4VP.

Verifiable credentials (VCs) are a verifiable form of digital credentials. They include, for example, digitized versions of passports, graduation certificates, and employee ID cards[*8]. The issuer digitally signs the credential, and they can be verified by third parties. Multiple VC standards exist, including ISO/IEC 18013-5 Mobile driving license (mDL)[*9] and W3C Verifiable Credentials[*10], which provides a general-purpose data format.

VCs are typically stored in an application called the credential holder's wallet. As mDL and W3C Verifiable Credentials are only VC data specifications, however, they do not define a protocol for obtaining credentials from an issuer and storing them in a wallet, nor a protocol for presenting credentials from a wallet to a verifier. The design of these protocols is up to the implementer. One example is SMART Health Cards (SHC)[*11], a specification for handling VCs (W3C Verifiable Credentials format) for medical information (incidentally, the Covid-19 vaccination certificates provided by Japan's Digital Agency are based on SHC[*12]). The OpenID Foundation is working to standardize these protocols in an effort to promote the adoption of VCs. This is in the form of OpenID for Verifiable Credential Issuance (abbreviated OID4VCI)[*13], a protocol for issuing VCs, and OID4VP, a protocol for presenting VCs. Both OID4VCI and OID4VP are independent of the VC data

*5 FAPI 2.0 Security Profile (https://openid.bitbucket.io/fapi/fapi-2_0-security-profile.html).
*6 FAPI: Client Initiated Backchannel Authentication Profile (https://bitbucket.org/openid/fapi/src/master/Financial_API_WD_CIBA.md).
*7 OpenID for Verifiable Presentations (https://openid.net/specs/openid-4-verifiable-presentations-1_0.html).
*8 Verifiable Credentials Use Cases (https://www.w3.org/TR/vc-use-cases/).
*9 ISO/IEC 18013-5:2021 —Personal identification —ISO-compliant driving licence —Part 5: Mobile driving licence (mDL) application (https://www.iso.org/standard/69084.html).
*10 Verifiable Credentials Data Model v1.1 (https://www.w3.org/TR/vc-data-model/).
*11 SMART Health Cards (https://smarthealth.cards/en/).
*12 Digital Agency, "FAQ: Contents of vaccination certificates" (https://www.digital.go.jp/policies/vaccinecert/faq_06/, in Japanese).
*13 OpenID for Verifiable Credential Issuance (https://openid.bitbucket.io/connect/openid-4-verifiable-credential-issuance-1_0.html).

specification and can be used with mDL, W3C Verifiable Credentials, or other formats.

So, we now turn to the main focus of this section, OID4VP. What does it mean to present a VC? Imagine a situation in which you are asked to provide age verification to purchase alcohol. With a physical ID, you have to present it face-to-face to a clerk at a brick-and-mortar store. VCs, on the other hand, are electronic data, so the interaction does not have to be face-to-face. You can use them for online shopping.

1. You put liquor in your cart on the liquor store website and click the purchase button. The site asks you to present a VC proving you are at least 20 years old.
2. When you click the submit button, your wallet is launched via a deep link, and a message asking if your VC can be presented to the liquor store is displayed.
3. If you consent, you are redirected back to the liquor store website, and your VC is passed to the liquor store. At this point, it is possible to use a mechanism called selective disclosure to ensure that the store only sees what it needs—your date of birth—and none of the other information in your VC.
4. The liquor store website verifies your VC, checks your age, and allows you to make the purchase if you are at least 20 years old.

The above is known as a same-device flow. This is when the software running OID4VP and the wallet are on the same device, that is, when inter-application redirects are possible. OID4VP also accommodates cross-device flows. In the previous example, this corresponds to the use of a VC stored in a smartphone wallet when shopping online on a PC. Instead of redirects, cross-device flows use QR codes to connect the two devices.

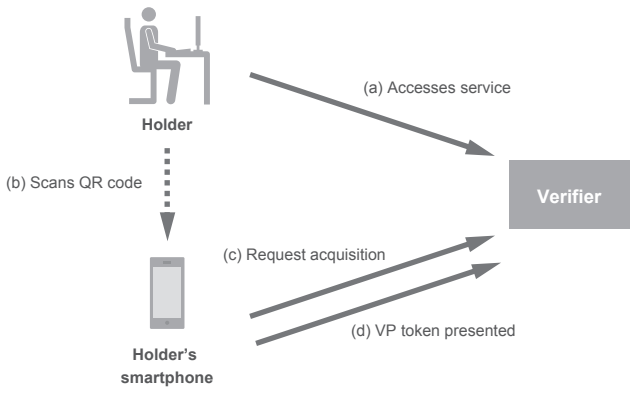


**Figure 3: Cross-Device Flow Authentication Example**

Let's take a closer look at OID4VP's cross-device flow (Figure 3). In OID4VP, the end user who has the VC is called the holder, the person to whom the VC is presented is called the verifier, and the data format used to present the VC to the verifier is called the VP token. A VP token can contain multiple VCs. The verifier's application needs a server that will receive HTTPS requests.

1. The holder accesses the verifier's services via a PC (a).
2. The verifier application converts the request acquisition URI into a QR code, which is displayed on screen.
3. The holder scans the QR code with a smartphone wallet (b).
4. The wallet accesses the verification server's request acquisition URI (c).
5. The verification server returns the details of the request in response. The request contains a detailed description of the requirements of the VC that will be presented.
6. In accord with the request received, the wallet displays a message asking the holder for consent regarding the content of the VC that will be presented.
7. The holder reviews the content and consents to the VC being presented.
8. The wallet sends the VP token to the verification server (d).
9. Once the verifier verifies the VC, the holder can continue to use the verifier's services via the PC.

With OID4VP's cross-device flow, all communication between the verifier and the wallet after the URI is initially acquired using a QR code is assumed to take place over the Internet. OpenID for Verifiable Presentations over BLE[*14] is an extension of this currently being developed to facilitate the use of OID4VP in environments where the

*14 OpenID for Verifiable Presentations over BLE (https://openid.bitbucket.io/connect/openid-4-verifiable-presentations-over-ble-1_0.html).

Internet is unavailable. Possible use cases for this include patrons presenting e-tickets in VC form wirelessly over BLE (Bluetooth Low Energy) at venues where smartphones are unable to establish a stable Internet connection, such as large concert venues or below-ground entertainment venues.

The potential use cases for VCs span all kinds of everyday scenarios. Once the OID4VP standardization process is complete and it truly starts to become widespread, we will no doubt encounter this cross-device flow in many aspects of our daily lives.

## 3.5 SIOPv2's Cross-Device Self-Issued OP

Self-Issued OpenID Provider v2[*15] (abbreviated SIOPv2) is a specification being developed by the OpenID Foundation. It extends OpenID Connect to allow end users to issue ID tokens themselves. The previous specification (SIOP sans v2) was part of the OpenID Connect Core 1.0[*16] specification, whereas SIOPv2 is now being standardized as an independent specification.

With OpenID Connect, an OpenID Provider (OP) issues an ID token that proves the end user's identity, and this is presented to any third party (the Relying Party (RP)) who wants to authenticate the end user. Social login (logging in via an account with Google, Apple, etc.) is a typical example of how this is used with web services. In these cases, Google or Apple or the like is the OP, and the web service is the RP. With SIOP, the end user acts as the OP and issues their own ID token.

The advantage of SIOP is that it allows end users to manage their own IDs, away from the mega platforms' centralized identity management. With social login, the OP is able to collect information on which RP was used. And if a user's OP account is suspended, this will also render the RP's service unavailable to that user. The idea of SSI (Self-Sovereign Identity) is beginning to gain traction as a means of overcoming these undesirable aspects of centralized identity management. The SIOP specification is designed to make OpenID Connect work with SSI.

The SIOPv2 protocol defines two flows. One is the conventional Same-Device Self-Issued OP, in which the RP client application and the OP run on the same device. Redirects are used to link the RP and OP. The other is Cross-Device Self-Issued OP, which was newly added in SIOPv2. Here, the OP runs on a different device (usually a smartphone). Let's take a look at the Cross-Device Self-Issued OP flow (Figure 4).

1. The end user accesses the RP (a).
2. The RP displays the self-issued request URI on screen, usually as a QR code.
3. The end user scans the QR code with a smartphone (b). The self-issued request URI is a deep link that launches the OP.
4. The OP is launched via the deep link. A message requesting permission to issue an ID token is displayed on screen.
5. Once the end user approves, the OP sends the issued ID token to the RP's backend server (c).

In addition to Cross-Device Self-Issued OP, SIOPv2 is expected to have the following enhancements over the previous specification.

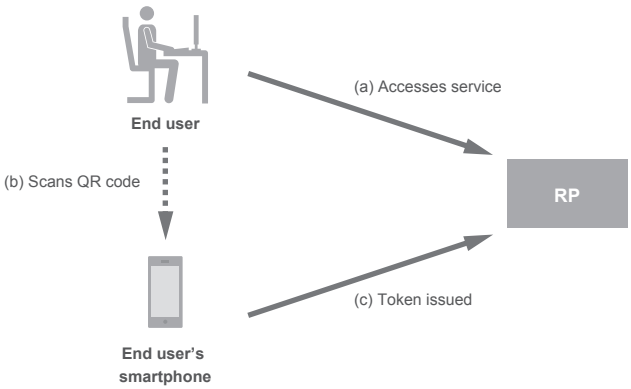


Figure 4: Example of Cross-Device Self-Issued OP Authentication

*15 Self-Issued OpenID Provider v2 - draft 12 (https://openid.bitbucket.io/connect/openid-connect-self-issued-v2-1_0.html).
*16 Final: OpenID Connect Core 1.0 incorporating errata set 1 (https://openid.net/specs/openid-connect-core-1_0.html).

- The end user's public key fingerprint is what has so far been used as the end user identifier included in the ID token. In addition to this, v2 will also allow the use of DIDs (Decentralized Identifiers). This will allow the use of an external verifiable data registry.
- When combined with OID4VP, it will allow VCs to be presented together with ID tokens. By verifying the VC, the RP will be able to associate the ID token with a VC issued by a trusted issuer. Since VC verification is completed on the RP (i.e., verifier) side, no information is collected by the VC issuer.

## 3.6 CTAP v2.2's Hybrid Transports

FIDO2[*17] is an authentication technology for passwordless sign-in to web services put forward by the FIDO Alliance. FIDO2 consists of W3C Web Authentication (WebAuthn)[*18] and corresponding Client to Authenticator Protocols (CTAP)[*19]. The WebAuthn specification is standardized by the W3C in collaboration with the FIDO Alliance. It is designed to facilitate web service sign-ins using biometric authentication entities, called authenticators, and authentication via security keys and the like. The CTAP specification is standardized by the FIDO Alliance. It is designed to allow the use of external authenticators that are not built in but instead connected to a device via USB or NFC.

CTAP v2.2, currently in the drafting phase, proposes a protocol called hybrid transports for using smartphones as external authenticators. In short, this would allow the use of a smartphone for authentication when signing in to a web service on a PC or the like. A number of operators already offer similar solutions, but they are all proprietary implementations. The FIDO Alliance is endeavoring to standardize the protocol. Authentication using hybrid transports will apparently be called FIDO Cross-Device Authentication flow (CDA)[*20]. Incidentally, there is also the somewhat similar sounding Multi-Device FIDO Credentials[*21]. This provides a mechanism for synchronizing credentials (authentication credentials) across an end user's own devices, and is also known as Passkeys[*22]. CDA and Passkeys are separate specifications, and hybrid transports can be used between PCs and smartphones even when credentials are not synchronized via Passkeys.

Figure 5 shows an example sign-in procedure using hybrid transports.

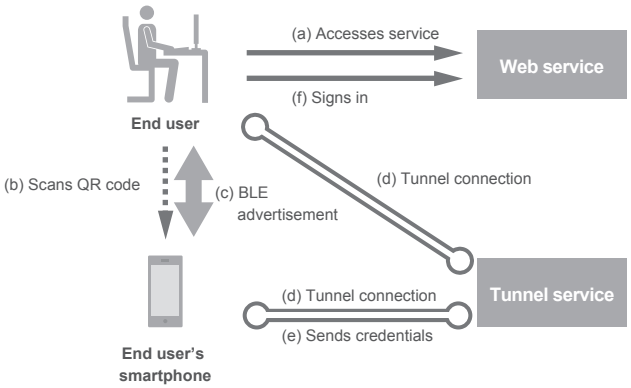


**Figure 5: Example of Sign-in Procedure using Hybrid Transports**

*17 User Authentication Specifications Overview - FIDO Alliance (https://fidoalliance.org/specifications/).
*18 Web Authentication: An API for accessing Public Key Credentials - Level 3 (https://www.w3.org/TR/webauthn-3/).
*19 Client to Authenticator Protocol (CTAP) (https://fidoalliance.org/specs/fido-v2.2-rd-20230321/fido-client-to-authenticator-protocol-v2.2- rd-20230321.html).
*20 Terms - passkeys.dev (https://passkeys.dev/docs/reference/terms/#cross-device-authentication-cda).
*21 White Paper: Multi-Device FIDO Credentials - FIDO Alliance (https://fidoalliance.org/white-paper-multi-device-fido-credentials/).
*22 Passkeys (Passkey Authentication) (https://fidoalliance.org/passkeys/).

1. Using a PC, the user opens the sign-in screen on a FIDO2-enabled website (a).
2. A dialog box for selecting the authenticator is displayed. The user selects “Smartphone”.
3. A QR code appears on screen.
4. The user scans the QR code using their smartphone (b).
5. The authentication application on the smartphone starts up.
6. To reduce the risk of phishing, at this point BLE advertisement is used to confirm that the PC and smartphone are in close proximity to each other (c).
7. The end user provides fingerprint or other authentication.
8. WebSocket is used to establish a reliable, secure communication link between the authentication application on the smartphone and the web browser on the PC (d). The tunnel specification is up to the implementer.
9. The authenticator application provides the credentials to the web browser through the tunnel (e).
10. The web browser uses the credentials to perform a WebAuthn sign-in (f).

Once the tunnel link is established, the QR code scanning step is skipped in subsequent authentications.

FIDO2 is specially designed to replace website sign-in procedures, so it can be used in combination with OAuth/OpenID Connect. Hence, it is expected that cross-device flows based on hybrid transports could be adopted for most of the areas covered by OAuth/OpenID Connect. While it is still in the drafting phase, the specification does have great potential when it goes into practical use.

## 3.7 Conclusion

In this chapter, I have introduced some cross-device flow specifications, both standardized ones and some still being drafted. Each has its own characteristics and target use cases. Yet they all use the features and functionality of smartphones (high penetration rate, always-on mobile, advanced biometric authentication, QR code support, push notification support, etc.) with the aim of providing safer, easier-to-use authentication and authorization flows. As cross-device flows become more prevalent, we can expect the security of online services and transactions to improve, providing an even better experience for users.

**Kenzo Yotsuya**
Research Laboratory, Internet Initiative Japan Inc.
Mr. Yotsuya is engaged in research and development on technologies related to next-generation authentication and authorization.