

Institute of Information Security

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Projekt-INF

# **Examining the Practical Security of the OAuth Device Authorization Grant**

Nicolai Krebs, Lynn Förster

<b>Course of Study:</b>	Informatik
<b>Examiner:</b>	Prof. Dr. Ralf Küsters
<b>Supervisor:</b>	Tim Würtele, M.Sc.

<b>Commenced:</b>	October 5, 2020
<b>Completed:</b>	April 6, 2021

## **Abstract**

The OAuth 2.0 Device Authorization Grant establishes the possibility of connecting IoT devices with limited input capabilities to accounts at various service providers. The simplified authorization process of the protocol, however, creates vulnerabilities that can be exploited by an attacker. We analyze the practical security of the Device Grant in a simulation environment. For this purpose, we implement multiple attack scenarios with a focus on the user perspective and evaluate their technical chances of success as well as the possibility of detection by the user.

Additionally, we included optional defensive measures and assess their effectiveness. Our simulation demonstrates that a number of attacks are feasible in practice if only the minimal requirements of the RFC specification is implemented. We argue that stricter requirements that will be presented in this paper can significantly improve the security of the Device Grant.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>The Device Authorization Grant</b>	<b>5</b>
2.1	OAuth 2.0 . . . . .	5
2.2	Flow of the Device Grant . . . . .	5
<b>3</b>	<b>Overview of the Simulation</b>	<b>7</b>
3.1	Default Scenario Without Attack . . . . .	7
<b>4</b>	<b>Attacks in the Simulation</b>	<b>10</b>
4.1	User Code Leak . . . . .	10
4.2	Device Code Leak . . . . .	13
4.3	Man-in-the-Middle . . . . .	15
4.4	Replay Attack . . . . .	16
4.5	Remote Phishing . . . . .	18
4.6	Cross-Site-Request-Forgery with QR-Code . . . . .	19
4.7	Corrupted Device Client . . . . .	21
4.8	Denial of Service . . . . .	22
<b>5</b>	<b>Evaluation of the RFC Specifications in the Context of our Simulation</b>	<b>23</b>
5.1	RFC 8628 - OAuth 2.0 Device Authorization Grant . . . . .	23
5.2	RFC 6749 - The OAuth 2.0 Authorization Framework . . . . .	24
<b>6</b>	<b>Real Implementations of Google etc.</b>	<b>25</b>
6.1	Google . . . . .	25
6.2	GitHub . . . . .	25
6.3	Amazon . . . . .	26
<b>7</b>	<b>Conclusion</b>	<b>27</b>
	<b>Bibliography</b>	<b>28</b>

# 1 Introduction

OAuth 2.0 is a widely established authorization protocol that grants a third-party service access to data and services of a user account provided by a so-called Identity Provider (IdP). With OAuth 2.0, popular services from an IdP, also known as resource provider, can be used in a larger range of applications. It also offers the option of connecting the account at the IdP to a third-party application, known as client, in a “Login with X” functionality, where X is the name of the IdP.

“Login with X” account services are advertised to application developers with advantages such as simplification in software development, granting access to the account’s resources without sharing user credentials, increasing user satisfaction, and security provided by the OAuth 2.0 protocol. Amazon claims that users prefer to use a “Login with Amazon” function over others, citing its practical aspects.

The Device Authorization Grant is an extension of the OAuth 2.0 protocol specifically designed for devices with limited input capabilities. In contrast to OAuth 2.0, the device client only displays a user code and a link to a login page of an Authorization Server (AS) that belongs to the IdP. The user opens the link in the browser of another device, e.g., a smartphone, and enters the user code and authenticates to authorize the device.

Since clients usually have access to sensitive data and other functionalities of an account at an IdP - often a major service provider that the user might be relying on for multiple services -, a flaw in the protocol could allow an attacker not only to extract and manipulate personal data, but also take control over devices.

The security of OAuth 2.0 has been formally proven under specific circumstances, among others by [FKS16] with the assumption of best practices in the implementation of the AS. In an informal security analysis, [KKK20] illustrated various potential vulnerabilities of the Device Grant given a minimal implementation of the RFC specifications.

It is our objective to analyze these threats by demonstrating various attack scenarios on the Device Grant in a simulation environment and evaluate the practical threat of the attacks described in [KKK20]. The simulation of the attacks illustrates that a user can perceive the effects of some parts of an attack while he may be entirely unaware of other aspects of an attack. In addition, various security relevant parameters can be configured and security measures enabled. We examine the effect of these parameters and measures and discuss whether the protocol specifications are sufficient to ensure maximum security in the context of our attack scenarios.

## 2 The Device Authorization Grant

### 2.1 OAuth 2.0

The OAuth 2.0 Authorization Framework specifies the authorization as a protocol between four entities. The *resource owner* is in most cases the user of the application, however, in some cases might be also be an application. The role of the *resource owner* is to grant a third-party application access to the resources of the account.

The *resource server* hosts the resources contained in the account and does not play a role in the process of authorization. It is responsible for providing the resources such as video streaming or an AI SmartHome assistant such as Amazon's Alexa service.

The authorization itself is handled by the *authorization server* (AS), responsible for verifying the user's account data and communicating the authorization codes to the device. It is hosted by the provider of the account.

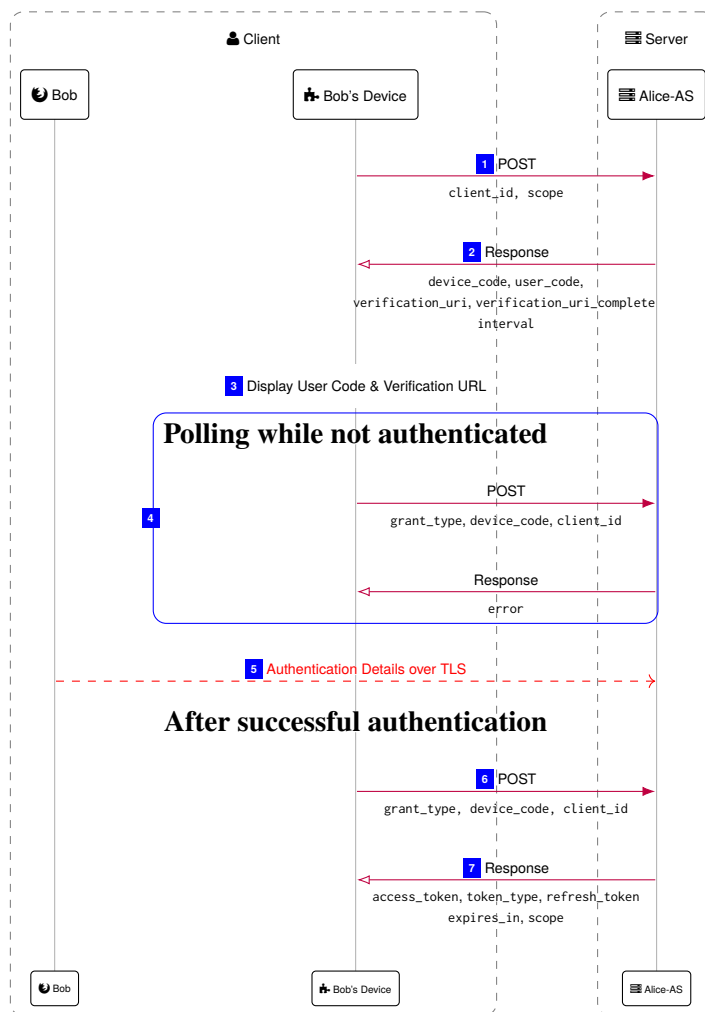
The last role is the *client*, the third-party application trying to gain access to the protected resources. In the general OAuth 2.0 framework, the client communicates with the AS and the *resource server* via a *user agent* such as a web browser enabling communication.

The main difference in the Device Grant is that the *user agent* is running on a separate device, not on the client device. In the Device Grant, the *client* is also called *device client*. We will use *client*, *device client*, and *device* interchangeably unless explicitly stated otherwise.

### 2.2 Flow of the Device Grant

If a user Bob wants to connect a device to his Alice-AS account, he starts the process on the device. The device sends a POST request to Alice-AS that specifies the `client_id` and optionally the scope of the Grant [1]. Alice-AS then generates valid authorization codes and responds by sending the `user_code`, `device_code`, and `verification_uri` to the device [2]. The response may also include an optional `verification_uri_complete`. Bob's device then displays the `user_code` and the `verification_uri`, prompting him to visit the `verification_uri` on another device to complete the authorization process [3]. Depending on the implementation, the device may also display the `device_code` to the user to confirm it against the `device_code` shown during login by Alice-AS, which is, however, not recommended.

The device begins to periodically poll for an `access_token` at Alice-AS [4] immediately after receiving the authorization codes. Polling requests include a `client_id` and `device_code`. Bob proceeds to visit the `verification_uri` of Alice-AS on another device and enters the `user_code`. If the device displays the `device_code`, Alice-AS asks Bob to confirm the code before completing the process by authenticating. Once he has entered valid account credentials [5], Alice-AS responds to the polling of the device with a valid `access_token` [7].



**Figure 2.1:** Message exchange of standard scenario of the Device Grant. Note that the graphics have been taken from [KKK20] with small modifications.

Devices with limited display space can also show the `verification_uri_complete` in a compressed form such as a QR code. Scanning the QR code then directly leads to the login page of the AS since the `user_code` is included in the URI.

## 3 Overview of the Simulation

The simulation is implemented in Python with the Web framework Django, version 3.1.3. For asynchronous tasking we use Celery, version 5.0.2, in combination with Redis, version 5.0.7 as a message broker.

In the frontend, the landing page includes a logger that gives insight to the main events happening during a given attack scenario and a simple user interface to configure various scenarios and attack vectors. The frontend of the client device is simulated in a separate browser tab and redirects the user to the login page of the authentication server. If the Device Grant is successful, a success message is shown in the client device user interface.

In the backend, the project is divided into 4 Django apps:

- configuration - handles logging and configuration of attack scenario and attack vectors.
- authorization\_server - includes a client registration endpoint for the client to obtain a `client_id`, a device authorization endpoint for the client device to initiate a Device Grant, a authentication or authorization endpoint for a user to login and authorize a device by means of a `user_code`, and a token endpoint for the client device to poll for an `access_token`.
- device - initiates the Device Grant and starts polling asynchronously for an `access_token`. It can be used both by the user and the attacker.
- attacker - comprises the main functionality of the attacks.

### 3.1 Default Scenario Without Attack

Figure 3.3 shows an overview of the implementation of the standard flow of the Device Grant. On completion of the attack configuration, the user is directed to proceed with the grant. The device user interface opens in a new browser tab and the user can initiate the grant [1]. In the following, all requests are POST request, unless stated otherwise.



**Figure 3.1:** User interface of the device

The device frontend then sends a request to the `device:grant` endpoint of the backend [2]. The device backend continues by sending a request for a `client_id` to `authorization_server` [3]. The device name is set to “Bob Inc. Streamingbox” and the scope is set to `LIMITED`. The device initiates the Device Grant by sending a request to the device authorization endpoint of `authorization_server` [5] and starts polling for an `access_token` by periodically sending a request to the token endpoint of the AS [7]. The user interface displays the response data including `user_code`, `verification_uri`, and `verification_uri_complete` [9] (see 3.1). The client frontend periodically checks for the success state of the grant [10].

The image shows a login form titled "Sign in". Below the title, it says "Use your AliceServices Account to connect your device with your account". Then, it states "You are connecting the following device: Bob Inc. Streamingbox". Below that, it says "You are granting your device LIMITED access to your account." There are two input fields: "Username" and "Password". At the bottom right, there is a blue button labeled "Confirm".

**Figure 3.2:** Login page of the AS with additional information

The links direct the user to the `user_code` entry page, if `verification_uri` is chosen, or directly to the login page, if `verification_uri_complete` is chosen [11]. The login page can be configured to show or hide the name of the device requesting access and the scope of the access granted (see 3.2). The AS responds with error messages in case of a wrong `user_code` or wrong login credentials. If the authentication is successful, a success message is shown.

If the grant is successful, the client obtains an `access_token` [13] and the user interface displays a success message [16]. If the user does not complete the authorization procedure or an error happens, polling will terminate after a timeout set in `configuration.models.py`.



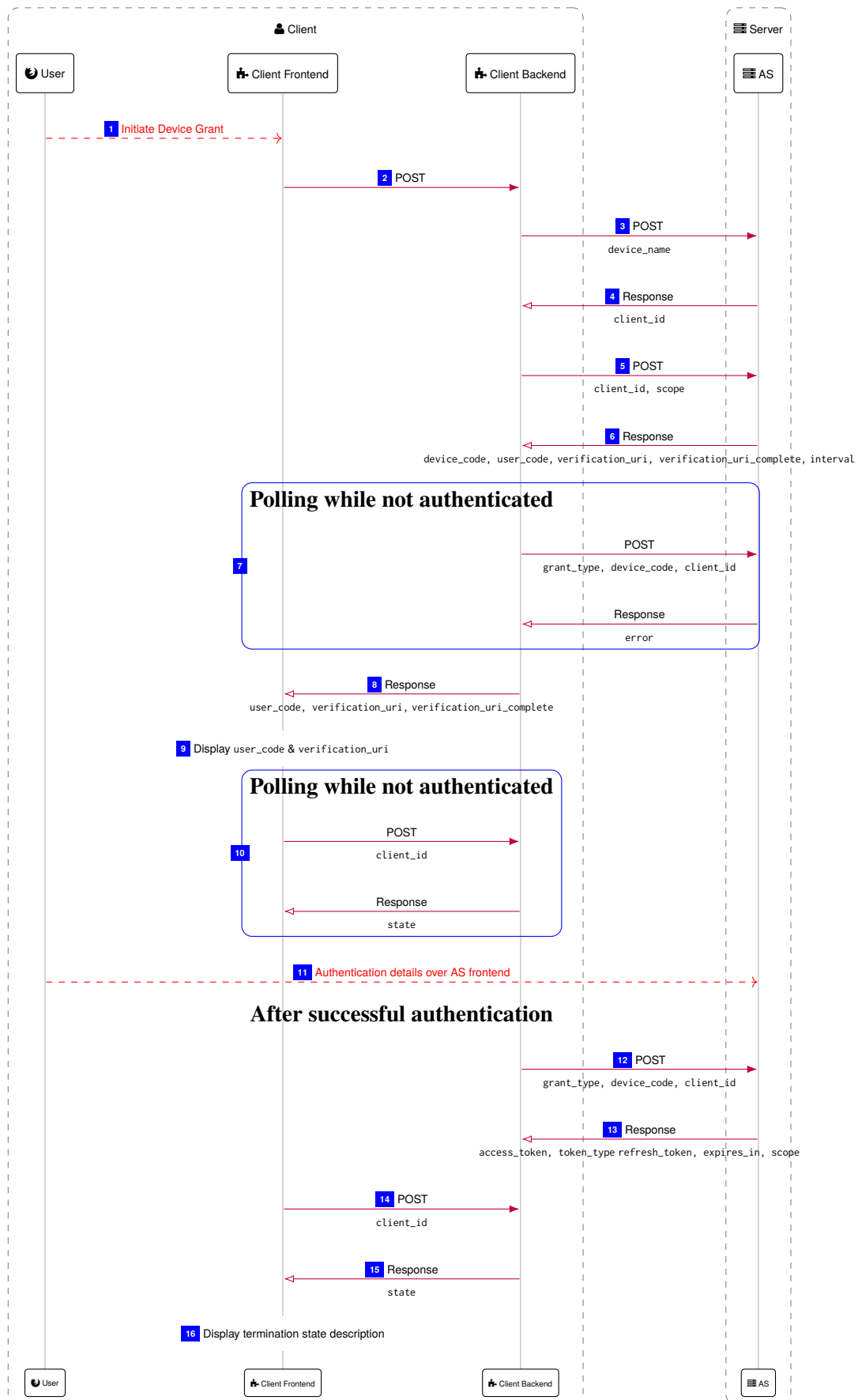


Figure 3.3: Message exchange of standard scenario of the Device Grant SEC Simulation

## 4 Attacks in the Simulation

### 4.1 User Code Leak

In the User Code Leak scenario Eve tries to intercept the `user_code` and hijack the login process in order to connect the Eve's account with Bob's device. Eve thus is able to spy on or may even control Bob's device.

#### 4.1.1 An Attack Scenario

Bob wants to register his SmartTV to his AliceTVs account. AliceTVs is the manufacturer of a TV that offers the option of connecting a device, such as a phone to the TV. With the phone, Bob can control the TV, e.g., turn it on and off. It also allows Bob to download apps to his TV. When Bob starts the authentication process, the `user_code` is displayed on the TV. Bob does not notice that Eve is able to see the TV screen through the window and observe the authentication process. Eve now enters the `user_code` herself and completes the authentication with her own account. Bob's TV shows a success message and he might not notice that the TV is now connected to Eve's AliceTVs account, although an error message is shown during the authentication process. Eve is now able to turn the TV on or off, regulate the sound, and install apps.

#### 4.1.2 Implementation in the Simulation

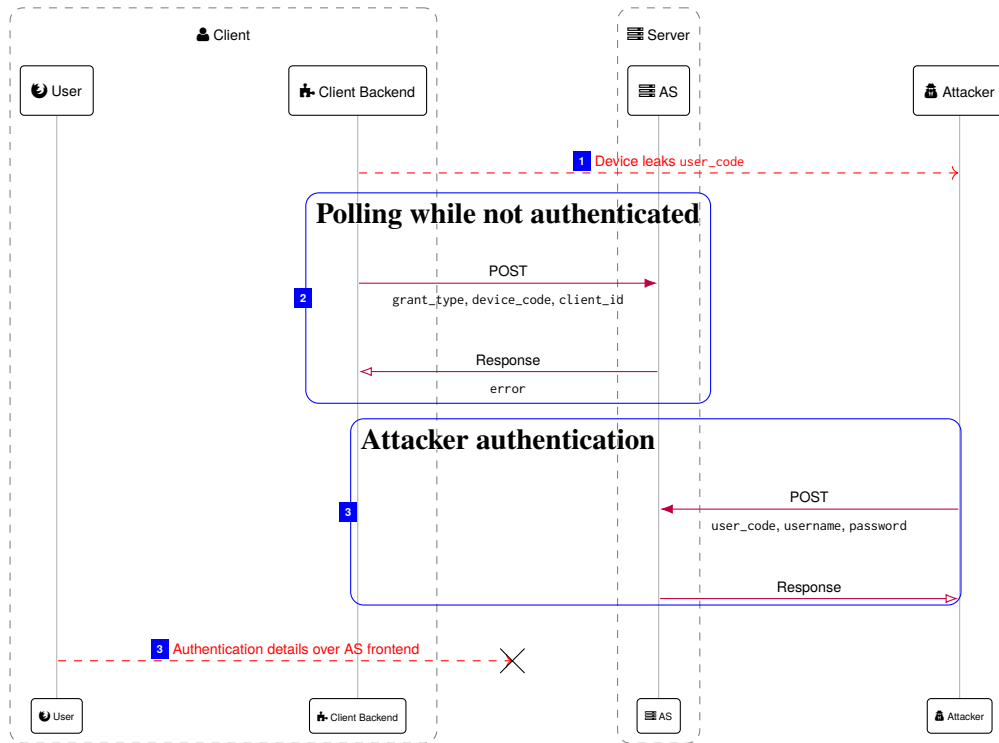
We distinguish between three sub-scenarios.

##### Attacker in Spatial Proximity

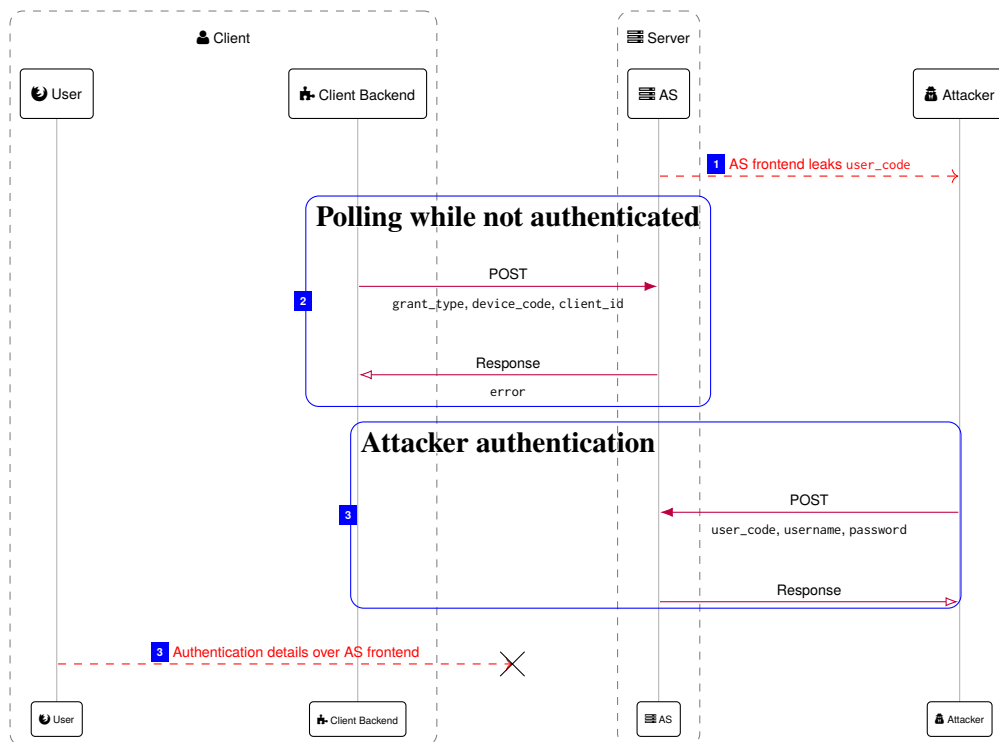
We simulate an attacker in spatial proximity who is able to see or in other ways come to know the `user_code` once it is communicated to the user. The device directly leaks the `user_code` to the attacker by sending a POST request containing the `user_code` [1]. The attacker then authenticates at the AS using his own account and the leaked `user_code` [3], the Device Grant is successful and the client device obtains an access token from the AS and shows a success message in its user interface. When the user attempts to log in at the AS with the `user_code`, an error message is shown, since the `user_code` is not valid anymore.

##### Attacker not in Spatial Proximity

The login page reached by calling `verification_uri_complete` shows an advertisement in an iframe that is provided by the attacker. When the page is loaded, a GET request is sent to the attacker [1]. The attacker extracts the Referer Header, i.e., `verification_uri_complete`, and thus obtains the `user_code` and follows the procedure described in first sub-scenario.



**Figure 4.1:** Flow of User Code Leak scenario with an attacker in spacial proximity

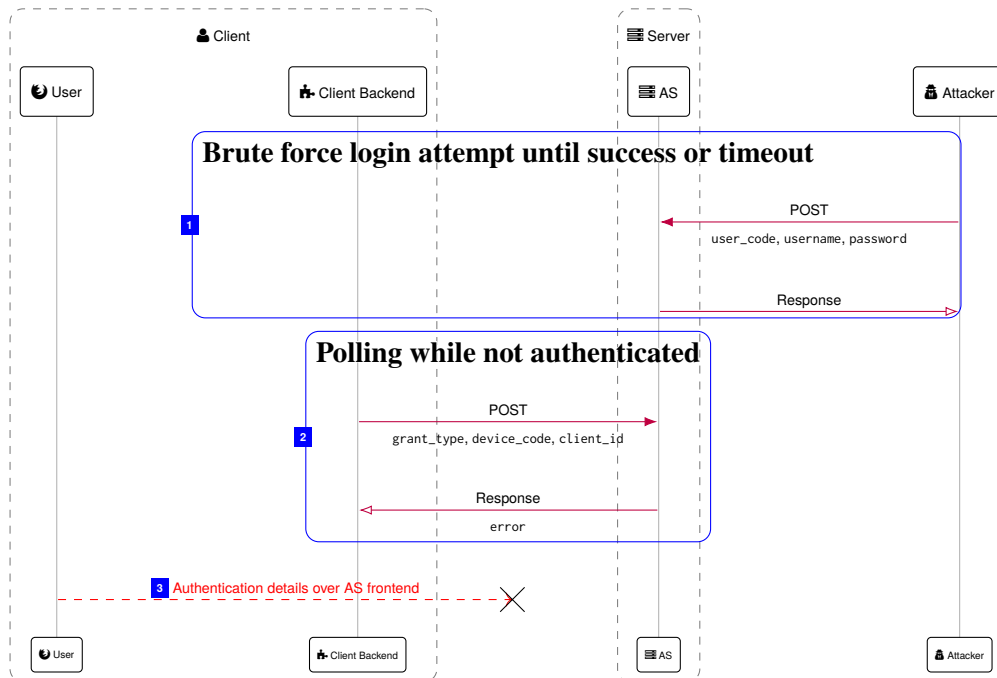


**Figure 4.2:** Flow of User Code Leak scenario with a spatially separated attacker

## Brute-Force Attack

The attacker asynchronously initiates a brute-force attack on the user\_code as soon as the configuration is complete. As there might be some delay between the completion of the configuration and the initiation of the Device Grant protocol, we repeatedly loop over the user\_code-domain and try to authenticate the attacker's user with the current counter until timeout [1]. For simplicity, both user\_code and device\_code are integers in range  $2^{\text{entropy}-1}$  to  $2^{\text{entropy}} - 1$ , where entropy is the number of bits used for encoding. entropy can be set to values between 2 and 20. During our test runs the attack was feasible up to an entropy of 12, however, the value may differ depending on the computational resources of the server the simulation is run on.

The success rate of the brute-force attack can be drastically lowered if the AS implements some kind of rate-limiting for authentication attempts. In the simulation, the AS counts and averages login requests per user and returns a 429 error once the rate per second is higher than 3. With rate-limiting, the attack was only feasible up to an entropy of 3 during our test runs.



**Figure 4.3:** Flow of User Code Leak brute-force scenario

### 4.1.3 Is the Attack Realistic?

The feasibility of the User Code Leak via spatial proximity is limited since the attacker would need physical access to a location where the user\_code is visible in the timespan of the authentication process. An attacker could gain knowledge of the user\_code in person or via camera surveillance. However, depending on the type of device and its display size and location, it may or may not be difficult.

The user can defend himself against the User Code Leak scenario in spatial proximity when informed about the attack principle and risks involved. For example, he could choose to authenticate his device only in a trusted environment or be very cautious in public settings.

Leaking the `user_code` via referer header is possible if an AS includes advertisements on the login page of the Device Grant. If the AS includes an advertisement owned by an attacker, the attacker would intercept `user_codes` by all users that visit the `verification_uri_complete` while his advertisement is active. He can then proceed to hijack the process on a potentially large number of devices. The attack can be easily prevented if the AS chooses not to include advertisements, however, the RFC specifications do not ban advertisements.

A brute-force attack on the `user_code` is a possibility. The entropy of a `user_code` is typically set at 8-10 alphabetic or numerical characters and separated by a hyphen for legibility, which still is convenient for the user and easy to remember. A `user_code` of length 8 has an entropy of  $26^8 \approx 2 * 10^{11}$ , assuming only upper- or lowercase characters are used, which would be another factor increasing convenience for the user and is for example implemented by Google and GitHub.

RFC 8628 describes a scenario with a `user_code` of base 20 and length 8, comparable to an alphabetic one. By itself, it would not reach the level of security of 128-bit symmetric encryption keys, which is assumed to be secure with current technology. The specification recommends rate-limiting as an additional protective measure. An AS that allows infinite attempts of authenticating the same device with changing `user_codes` is vulnerable to the User Code Leak. Rate-limiting is easy to implement in case of the User Code Leak since the attacker constantly tries to authenticate the same device with the same `device_code`. Hence, a User Code Leak can be prevented in practice, however, it depends on the AS's implementation of the RFC's recommendations.

## 4.2 Device Code Leak

Although the authors of [RFC862819] presume that the `device_code` is not displayed to the user (Section 3.3 and 5.2), they recommend not to show it for practical reasons, i.e., not to confuse the user, and not for security-critical reasons. Therefore, it is possible that an app developer implements the `device_code` to be shown on the client device.

Here we assume a weak implementation of the RFC specifications, showing the `device_code` to the user for comparison. If the `device_code` is displayed to the user, similar attack scenarios as in the User Code Leak are possible. An attacker might gain knowledge of the `device_code` in spatial proximity, via a referer header, or a brute-force attack. If the `device_code` is not shown to the user, only the brute-force attack would be feasible.

### 4.2.1 An Attack Scenario

Bob tries to connect his TV with his AliceTVs account as in section 4.1.1, with the difference that the TV shows a `user_code` as well as a `device_code` during the registration. If Eve can now obtain the `user_code`, she can enter it on the AliceTVs registration site, leading to the site that asks her to check if the `device_code` is correct. She therefore can obtain the `device_code` if she already has the `user_code`. With the `device_code` Eve can now proceed to start polling for an `access_token` at AliceTVs since the `client_id` is public and she is in possession of the `device_code`. Bob enters his account credentials and Eve obtains an `access_token` from the AS, allowing her to register her own device with Bob's account. Bob sees a success message even though his TV is not registered. If the message is vague, he might assume that the Device Grant has not been successful instead of noticing that another device is now connected to his account.

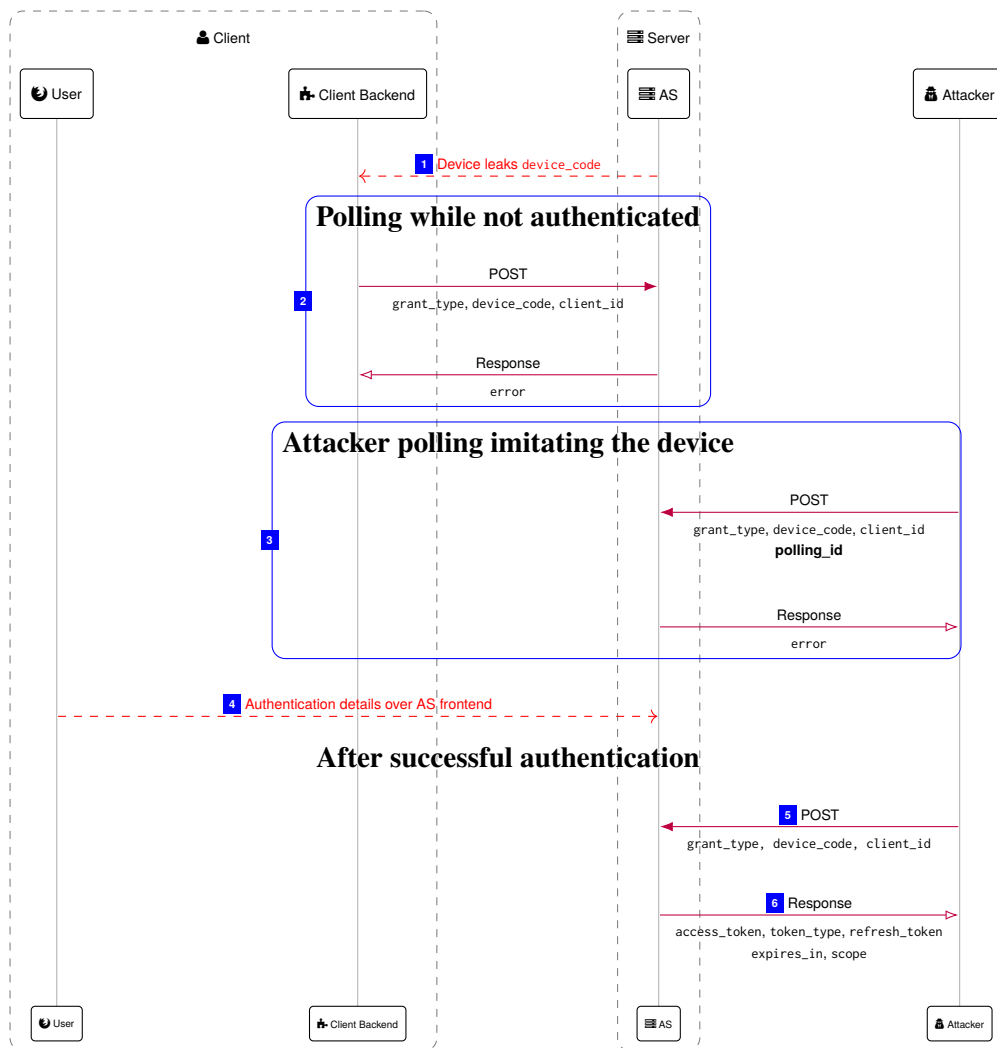
### 4.2.2 Implementation in the Simulation

We again distinguish between three sub-scenarios.

#### Attacker in Spatial Proximity

We simulate an attacker in spatial proximity who is able to see or in other ways learn the device\_code. We assume public clients, hence, the device's client\_id is publically available. Again, the device directly leaks the device\_code to the attacker by sending a POST request containing the device\_code [1]. For simplicity, the device also includes the client\_id in the same request. The attacker then starts polling for an access\_token asynchronously until a user successfully authorizes the client device or a timeout is reached [3].

The AS does not check frequency of token endpoint requests and the attacker does not respond to slow\_down responses, hence, the polling frequency of the attacker is much higher compared to that of the client device and the attacker has a high probability of succeeding in his attack.



**Figure 4.4:** Flow of Device Code Leak scenario

### Attacker not in Spatial Proximity

The first part of the attack for a spatially separated attacker is identical to the scenario described in Section 4.1.2. The attacker obtains the `user_code` from the Referer Header of the GET request by his advertisement on the login page of the AS. In the scenario, the AS shows the `device_code` on the login page after user code entry, which the attacker is now able to extract as he has gained knowledge of the `user_code`. The attacker request the login page from the AS, extracts the `device_code`, and sends a POST request to the AS to find the matching publically available `client_id` belonging to the `device_code`. He starts polling and obtains a valid `access_token` once the user is authenticated.

### Brute-Force Attack

In the brute-force scenario the attacker asynchronously awaits new client registrations. Once a new `client_id` is issued, he starts polling for `client_id`, `device_code` combinations by looping over the `device_code` domain [3] (see Section 4.1.2 for more details). In the case of an invalid `device_code`, the AS responds with an `invalid_client` error message and the attacker continues to poll for a different combination. When a valid combination is found, the AS responds with `authorization_pending` and the attacker continues polling and the attack concludes exactly like the previous sub-scenario.

As in the User Code Leak scenario an effective protection mechanism is rate-limiting. In the Device Code Leak scenario, we identify the attacker devices by an additional POST parameter that is included in token endpoint requests (see highlighted parameter in [3]). As described in Section 4.1.2, the AS measures the request rates of the attacker device and responds with a 429 error when the configured rate-limit is exceeded.

#### 4.2.3 Is the Attack Realistic?

If the `device_code` is not shown to the user, there is no limit on the entropy for reasons of user convenience. As such, `device_codes` can and should be chosen with significantly higher entropy than `user_codes`, rendering a brute-force attack in the Device Code Leak more impractical than for the User Code Leak scenario.

If the `device_code` is shown to the user, the Device Code Leak and the User Code Leak have similar chances of success. From a technical standpoint, polling faster than the user's device is possible and likely to succeed regardless of rate-limiting constraints by the AS. The user might be sceptical of a success message when his device is not registered. However, he can execute the Grant again and try to register his device again after the attacker's device has already been registered. This is not possible in the User Code Leak, where the device indicates that it is registered.

With the requirements of not showing the `device_code` to the user and a sufficiently large entropy of the `device_code` to ensure security in practice, the Device Code Leak can be effectively prevented. In addition, an IdP could improve security by notifying the user about new devices connected to his account.

## 4.3 Man-in-the-Middle

In a Man-in-the-Middle (MITM) attack, the attacker establishes himself as a third-party in the communication between the device and the AS. He can intercept and manipulate requests of the device, forward them to the AS, and intercept responses of the AS in order to gain access to valid authorization codes.

### 4.3.1 An Attack Scenario

In a successful MITM attack, the user does not notice any difference to a normal iteration of the Device Grant. Bob starts the device grant on his device, proceeds to authenticate successfully at AliceMovies, and his device confirms that it is now connected with his account. Eve has the option of not forwarding the `access_token` to the device, in which case the Device Grant is not be completed and the device would most likely show an error message after timeout.

### 4.3.2 Implementation in the Simulation

For simulation purposes in this scenario the AS base address known to the client device is changed to the attacker's base address. All endpoints needed for the communication between client device and AS are emulated by attacker. Once the user initiates the Device Grant, the client device now sends all requests to the attacker, the attacker intercepts the requests, extracts relevant information, and forwards the request to the AS while impersonating the client device. In particular, the attacker is able to escalate privileges by setting the scope parameter in the device authorization request to `FULL` as opposed to the client device's specification of `LIMITED` [6]. Once the user has successfully authorized the device at the AS, the attacker intercepts a successful `access_token` response [13].

If the attacker was configured to forward the `access_token` response, the Device Grant concludes with a success message in the device user interface, if not, a timeout error message is displayed [14].

### 4.3.3 Is the Attack Realistic?

The RFC specifications require the use of TLS with server authentication as well as the validation of the AS's TLS certificate by the client [RFC674912]. Under these assumptions, it is unlikely for the attacker to successfully pose as a man-in-the-middle. Yet, as pointed out in [KKK20], there are vulnerabilities that if exploited to impair the device's ability to verify certificates. Such vulnerabilities can be leveraged by an attacker to attempt a MITM attack. The protocol targets a wide range of IoT (Internet of Things) devices and it has to be assumed that quality control for IoT devices is inadequate in many cases. Therefore, we cannot exclude the possibility of a successful MITM attack on the Device Grant.

## 4.4 Replay Attack

While researching the different attacks on the OAuth 2.0 protocol, we came to the conclusion that a replay attack as described in [KKK20] is not practically feasible.

In the scenario, an attacker can extend a device's authorization in the Grant beyond the lifetime of an `access_token` by repeatedly identifying and replaying an `access_token` request sent to the AS from the device. The AS does not include a mechanism to detect a successful completion of the Device Grant and thus would continue to issue new `access_tokens`.

In [RFC862819], however, the assumption is made that the user communicates with the AS over a TLS-encrypted session. Additionally, all requests from the device "MUST" use the TLS protocol and implement best practices. TLS offers protection against Replay attacks by using a different key for every new connection and a sequence number for every packet [RFC224699]. Assuming that the key exchange between the device and the AS is not intercepted or modified by the attacker, the attacker is not capable of encrypting messages with changed content. If he intercepts the message that requests a new `access_token`



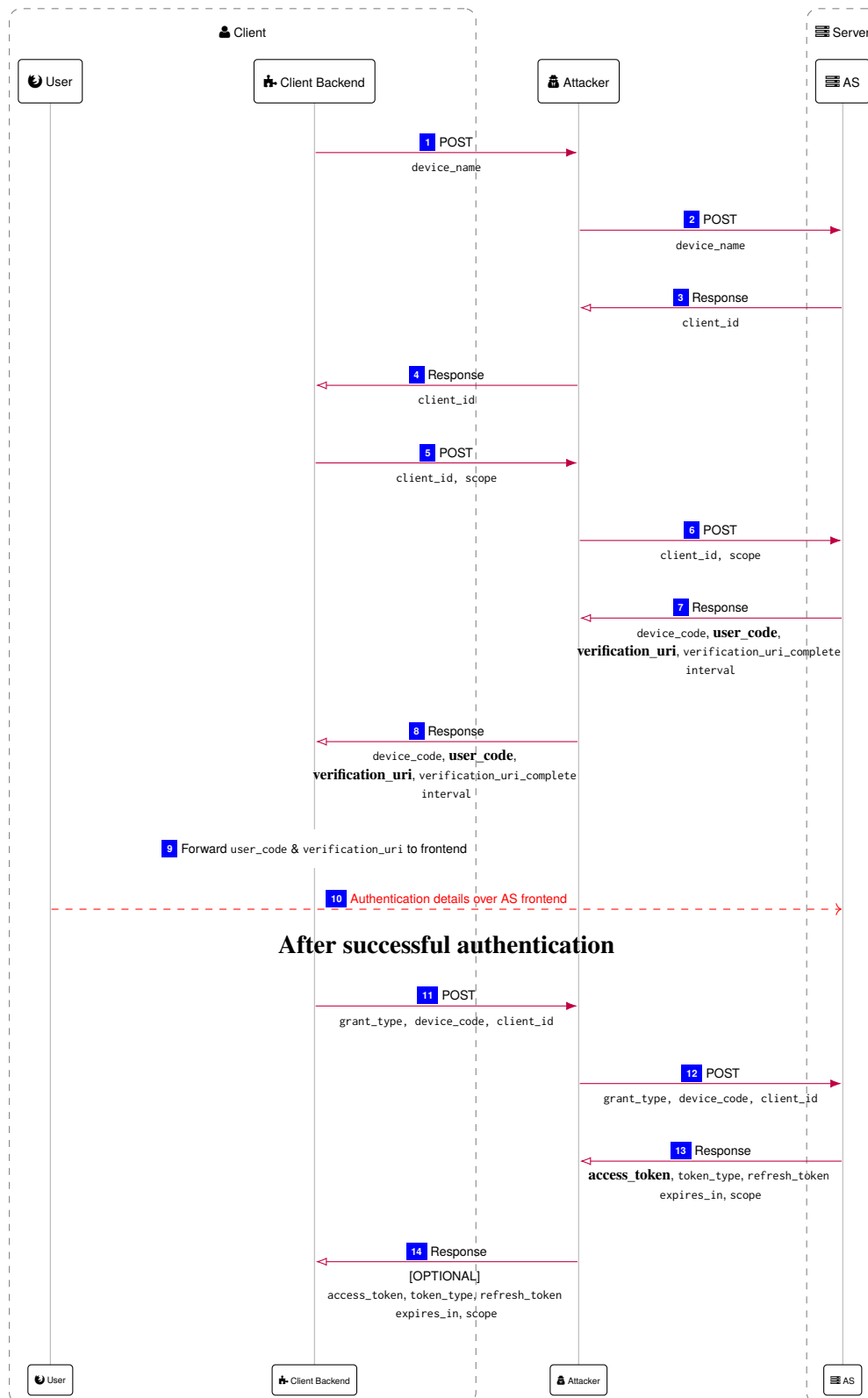


Figure 4.5: Flow of Man-in-the-Middle scenario

and identifies it as such, he can try to resend it to the AS. However, the message would be recognized as duplicate or invalid and be discarded. Hence the attacker is not able to trick the AS into issuing a new `access_token`. Hence, the replay attack as such is not feasible.

## 4.5 Remote Phishing

A remote phishing attack on the Device Grant works by sending the user a link via email for example. The email could contain either a link to a site showing the `verification_uri_complete` for the Device Grant or lead to a site that shows the user the `user_code` for authentication, in turn forwarding him to the authentication page of the AS for authentication. The user thus connects a device with his account without knowing it, leading to a security risk for his personal data or potentially authorizing the attacker to make purchases on his account.

### 4.5.1 An Attack Scenario

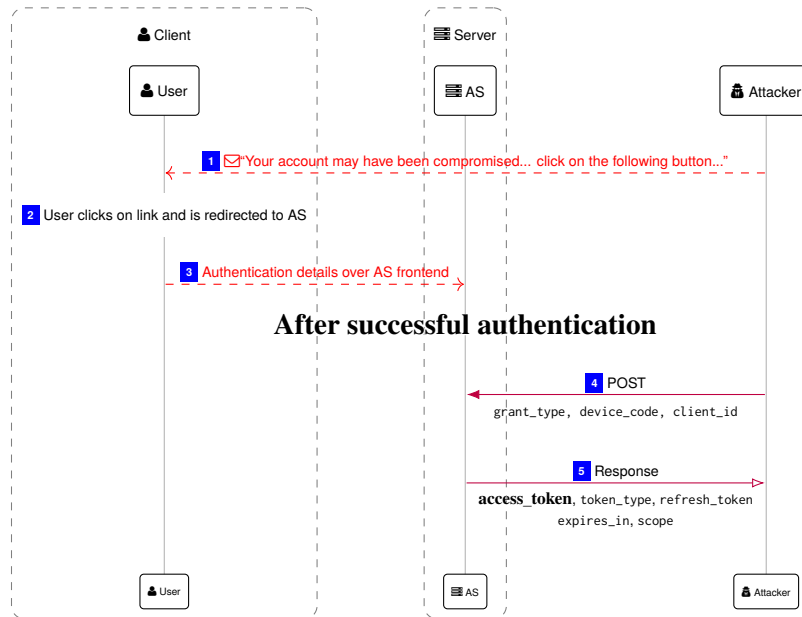
Bob receives an email with the subject “Your AliceMovies Account is compromised”. The email address of the sender appears legitimate and he might not be aware that an attacker can spoof email addresses. Therefore, he decides to trust the email. In the email body he is informed that his account was hacked and that he needs to enter a security code to verify it. The email contains a link for verification and Bob clicks on it. The link leads to a site that is designed in the typical AliceMovies design and that contains a `user_code` or “security code”, prompting him to click on another link to enter the code and log in to his account to verify it. Bob follows the instructions, unaware that he just registered Eve’s device to his account. The AS shows a message such as “Authorization successfully completed” and Bob is relieved that his account is secure again. Eve is now able to purchase movies with Bob’s account and potentially access to sensitive information such as payment details.

### 4.5.2 Implementation in the Simulation

The attacker initiates the Device Grant by sending a POST request to the grant endpoint of device and receives a response with a `user_code` and `verification_uri`. The device name is set to “Charlie Ltd. Smart Refrigerator” to indicate ownership by the attacker. We simulate the phishing attack by displaying a mock phishing email in a new browser tab [1]. The user is urged to log in to his account at the AS by clicking on a button [2]. Depending on the configuration, a click on the button either opens `verification_uri_complete` in a new tab or directs the user to a phishing page on Eve’s server that shows a new `user_code` and `verification_uri`. In the former case, the device grant is initiated before the phishing mail is created, in the latter case, it is initiated after the user clicks on the button directing to Eve’s server.

### 4.5.3 Is the Attack Realistic?

Phishing is one of the most popular attacks in practice. Whether an attacker manages to make a user click on the link depends on how credible the crafted email looks. While phishing is unlikely to succeed against an attentive user who is aware of the threat, there are many users who do not have sufficient knowledge of phishing. As a result, phishing is often implemented on a large scale, targeting many users at once to increase the chances of success.



**Figure 4.6:** Flow of Remote Phishing scenario

The option of embedding a QR code that contains the `verification_uri_complete` or including the URI itself simplifies the attack significantly. The RFC specifications suggest a limited lifespan of the `user_code` for protection against phishing (Section 5.4 in [RFC862819]) although a limited lifetime mainly protects from embedding the `verification_uri_complete` directly in the email. If the link in the email leads to a website owned by the attacker that, in turn, contains the `verification_uri_complete`, the attacker can obtain a new `user_code` from the AS once a user opens the website. However, the user might be suspicious of the uri of the attacker's website as it does not exactly match the IdP's address.

Phishing remains a relatively simple attack from a technical standpoint and needs to be taken into account by software developers who employ the Device Grant.

## 4.6 Cross-Site-Request-Forgery with QR-Code

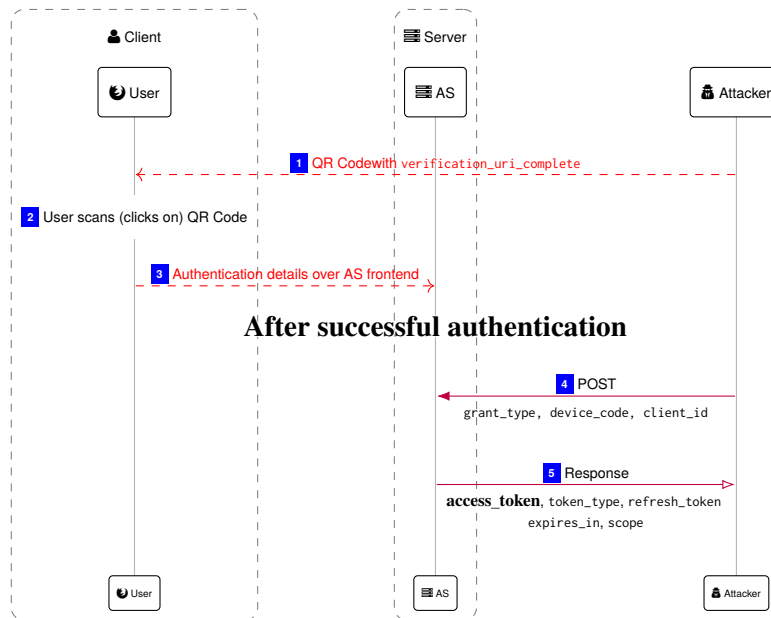
This attack utilizes the `verification_uri_complete` in the form of a QR code to trick users into performing an authorization against their will. Depending on the lifetime of the `user_code` and the information provided by the AS on the login page, the QR code can be distributed in physical form such as flyers or digitally in advertisements.

### 4.6.1 An Attack Scenario

Bob is browsing a website when she notices an advertisement banner that informs him that he is eligible for participation in a lottery. All he has to do is to scan the QR code and login with his Alice-AS account in order to participate.

### 4.6.2 Implementation in the Simulation

The attack works analogously to Remote Phishing. Instead of a mock email, the attacker computes a QR-Code of “verification\_uri\_complete” that is displayed to the user [1]. The user can click on the QR-Code and is forwarded to the login page of the AS [2].



**Figure 4.7:** Flow of CSRF with QR-Code scenario

### 4.6.3 Is the Attack Realistic?

Since the lifetime of the user\_code is only recommended to have a limit, it is possible that an AS would implement them to be valid for an unlimited time. The attacker could then print out the QR code and paste them onto advertisements such as client satisfaction stickers in trains asking for feedback about the service. Some QR code scanners do not show the URL of the site the QR code leads to, and if they show the URL, the user might trust the URL since it belongs to Alice-AS.

The chances of the CSRF attack succeeding depend on the AS asking the user whether he wants to connect a device to his account. If this question is not implemented, the user can mistake the login form for a login page that leads to his account details, not granting permission to a device authorization.

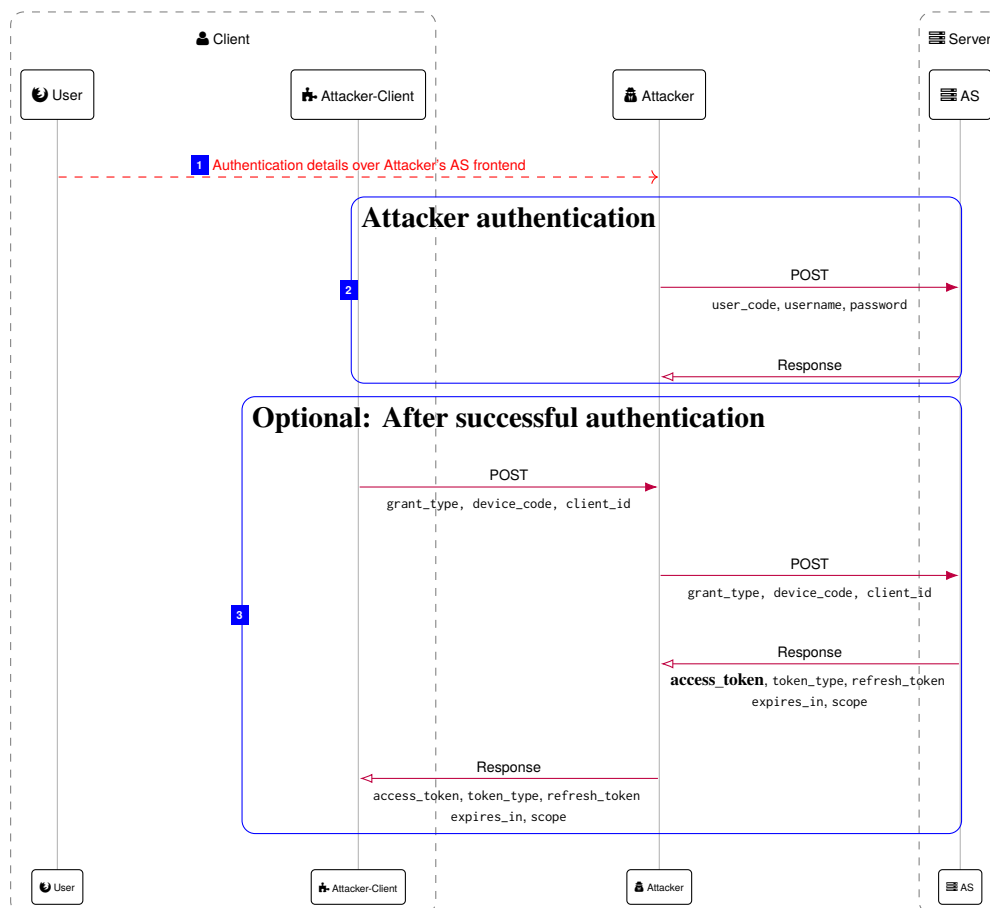
This attack can be effectively defended against by implementing a limited lifespan of the user\_code as well as an additional question to confirm that the user is aware of the device registration. Limiting the lifetime of the user\_code itself does not mitigate the risk of digital advertisements that can update the QR codes regularly.

## 4.7 Corrupted Device Client

A corrupted device client is dangerous from multiple perspectives, since it has access to the service data. For example a security camera that is not trustworthy could stream its video feed directly to an attacker. However, it can also trick the a user into disclosing his account credentials through the Device Grant by displaying a forged `verification_uri` that leads to a forged AS site. If the user enters his authentication details, the attacker can gain full access his account.

### 4.7.1 Implementation in the Simulation

The simulation works similar to the Man-in-the-Middle scenario, i.e., the attacker can intercept all traffic between the device and the AS. In addition, we set `verification_uri` and `verification_uri_complete` in `device:grant` to the uri of the attacker's authentication endpoint. Hence, `user_code` entry and login takes place on the attacker's app that simulates the attacker's server [1]. The attacker checks the validity of the `user_code` and login credentials by forwarding the login request to the AS [2] and returns the appropriate error or success message. If the attacker is configured to complete the Device Flow, the attacker then authenticates the user at `authentication_server` [3]. Else, the corrupted client device does not poll for an access token and will show an error message after timeout.



**Figure 4.8:** Flow of Corrupted Device Client scenario

#### 4.7.2 Is the Attack Realistic?

Since the Device Grant assumes a trustworthy device, this attack scenario has no concrete countermeasures specified in the RFC specifications. The production of untrustworthy device clients, however, is possible and leaves the user vulnerable to multiple security risks, among them the Corrupted Device Client.

### 4.8 Denial of Service

A DoS attack differs from the other attacks analyzed in this paper by its objective of temporarily blocking the authorization service. There is no risk of user data being stolen or accounts used for purposes not intended by the account owner, instead it violates the security characteristic of service availability. The attacker achieves this unavailability of service by starting the Device Grant on many devices controlled by the attacker, such as a botnet, at the same time, spamming the AS server with requests. The AS has to store the state of the started protocol and the attacker can exhaust its capacity.

#### 4.8.1 Implementation in the Simulation

In addition to the normal flow of the Device Grant (see 3.1), the attacker asynchronously initiates the Device Grant on 20 devices with name “Charlie Ltd. Smart Refrigerator” by sending POST requests to the grant endpoint of device. The devices start polling in concurrently, and the `authorization_server` measures the request rate to the `token_endpoint`. If the request rate surpasses a limit set in `configuration.models.py`, the `authorization_server` returns an HTTP response with status code 503 to all subsequent requests to any of its endpoints. For simulation purposes, we set the limit to 3 requests per second. The user device shows an appropriate error message.

#### 4.8.2 Is the Attack Realistic?

The Device Grant is a protocol that requires the AS to store the status of open Device Grants. As such, an overload of requests is possible since the server capacity of the AS is limited. The protocol does not offer specific methods of identifying and blocking an attacker that is intentionally trying to attack the AS by flooding it with requests. While polling with specific `device_codes` can be blocked, an attacker that controls a large botnet of IoT devices can easily bypass this limitation by registering multiple devices simultaneously. Although various defense techniques against DoS attacks exist, DoS remains a viable attack, in particular, as IoT devices gain more and more popularity.

## 5 Evaluation of the RFC Specifications in the Context of our Simulation

### 5.1 RFC 8628 - OAuth 2.0 Device Authorization Grant

The RFC specifications do not mandate that the `user_code` is a unique value. A scenario where two devices are assigned the same `user_code` for the Device Grant is unlikely but possible, in particular, as the entropy of the `user_code` is limited. If one of the users enters the `user_code` on the site of the AS, the AS could not identify which device the user wants to authorize. Since the behavior of an AS in this case may be arbitrary due to a lack of specification and therefore could result in a server error. The RFC therefore should demand uniqueness of the `user_code`.

The simulation gives insight into the threat of using Referer Headers combined with advertisement on the login page of the AS in order to obtain a `user_code` and connect a user's device with his own account. The RFC should prohibit the display of advertisement during the login process as a safeguard for this attack or make the developer aware of its risks.

The specification recommends the implementation of rate-limiting to prevent User Code Leak brute-force attacks [RFC862819]. Given that the entropy of the `user_code` is chosen as a compromise between user convenience and security and is therefore limited, the `user_code` is more vulnerable to brute-force attacks than the `device_code`. The authors of [RFC862819] consider the example of an 8 character code with base 20 secure when combined with a limit of five attempts. As discussed in 6, the example presents a realistic entropy for `user_codes` in practice. However, the implementation of rate-limiting is not transparent. If an AS does not implement rate-limiting, the `user_code` does not fulfill the criterion of being as difficult to guess as a  $2^{128}$ -bit encryption key.

We consider it security-critical not to display the `device_code` to the user for two reasons. Firstly, it significantly increases the chances of successful Device Code Leak attacks either by means of an interposed User Code Leak or in spatial proximity. Second, it imposes a limit on the entropy of the `device_code` if the user is expected to compare the codes. A lower entropy makes the `device_code` more vulnerable to brute-force attacks. A recommendation for the minimum entropy of the `device_code` should be included instead of citing that it "SHOULD" be "very high" [RFC862819].

The goal of displaying the `device_code` to the user is to offer him the chance of verifying which device he is connecting to his account. We consider it good practice to include this additional security measure without showing the `device_code` to the user. Instead, the AS should show the name of the device to the user and include a question such as "Are you sure that you want to connect *Bob's TV* to your Alice-AS account?". The RFC should incorporate this recommendation as a requirement for additional security as it increases transparency of the process to the user and thereby could reduce the possibilities of successful Device Code Leaks as well as Remote Phishing attempts. Additionally, the scope of the requested access should also be included, if possible, to increase the awareness of the user. In some of the attacks mentioned, the attacker is able to escalate the privilege of the `access_token` during the process, gaining greater access rights to the user's account than intended.

## 5 Evaluation of the RFC Specifications in the Context of our Simulation

[RFC862819] also recommends that the lifetime of the `user_code` should be “sufficiently short”. The lifetime is a major factor in a brute-force attack on the `user_code` as well as some forms of Remote Phishing and the CSRF attack.

All the previously mentioned recommendations made in RFC8628 are specified as “SHOULD” or “RECOMMENDED”. The definition of these keywords according to [RFC211997] is that “there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.” If only the minimum requirements of the specifications are implemented, the Device Grant is vulnerable to many attacks.

### 5.2 RFC 6749 - The OAuth 2.0 Authorization Framework

[RFC674912] specifies the OAuth 2.0 Authorization Framework and lays the foundation for the Device Grant. It also considers prevention or mitigation measures of possible attacks on the OAuth 2.0 protocol that are relevant to the Device Grant.

It specifies a maximum probability for an attacker to guess generated tokens such as the `access_token` at  $2^{-128}$ . However, there is no such value specified for the `device_code`. As we have pointed out, there is no need to show the `device_code` to the user. We consider this a flaw in the specifications and that may lead to a violation of the following requirement:

“The authorization server MUST prevent attackers from guessing access tokens, authorization codes, refresh tokens, resource owner passwords, and client credentials.” [RFC674912]

If this requirement is fulfilled, many of the attacks would not be possible. Since a knowledge of a `device_code` in combination with public clients enables an attacker to obtain an `access_tokens`, [RFC862819] should demand a constraint on the entropy of the `device_code`.

Additionally, in [RFC674912] the device is assumed to be trustworthy and capable of checking certificates and communicate over TLS. In practice, as discussed in Section 4.3.3, this assumption can be violated. Software flaws may exist that could compromise the device’s ability to check certificates and thus compromise the secure communication between device and AS. Under these assumptions, the Device Grant remains a protocol with vulnerabilities in practical implementations that can be exploited by an attacker, especially when an ignorant and unattentive user is involved.

[RFC674912] requires the use of TLS for the communication between AS and user. In this context, TLS provides security of the authentication process. However, an attacker is still able to perform phishing attacks that direct the user to the verification site of the AS and trick the user into completing the Device Grant 4.5.



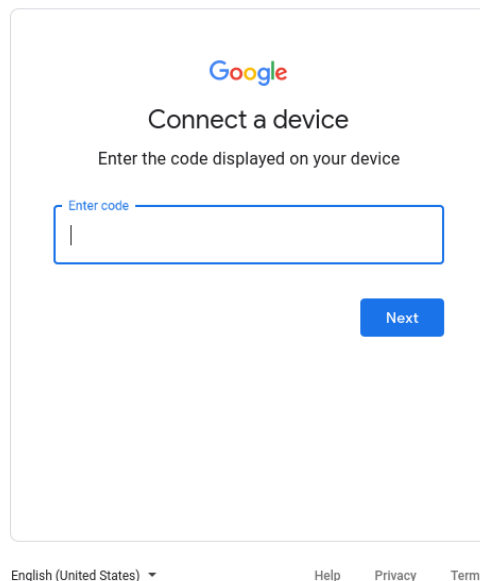
## 6 Real Implementations of Google etc.

### 6.1 Google

Google shows an example user\_code and device\_code on their application developer website [GOOGLE]. The entropy of the example device\_code of 46 US-ASCII characters is larger than  $90^{46} \approx 2^{298}$ , making a brute-force attack infeasible. Google is also implementing the recommendation of not displaying the device\_code to the user.

The user\_code can “contain any printable character from the US-ASCII character set”, according to Google [GOOGLE]. The example shown on the developer website shows an 8 character user\_code, assuming that it is separated by a hyphen for readability, resulting in an entropy of more than  $90^8 \approx 2^{51}$  which is similarly more than sufficient to protect from a brute-force attack.

Google also employs the recommended measures against phishing attacks during the authentication process. The user is clearly informed about the registration of a device.

The image shows a web interface for connecting a device to Google. At the top is the Google logo. Below it, the text 'Connect a device' is centered, followed by the instruction 'Enter the code displayed on your device'. There is a text input field with a blue border and a placeholder 'Enter code'. To the right of the input field is a blue button labeled 'Next'. At the bottom of the page, there is a language selector 'English (United States)' with a dropdown arrow, and links for 'Help', 'Privacy', and 'Terms'.

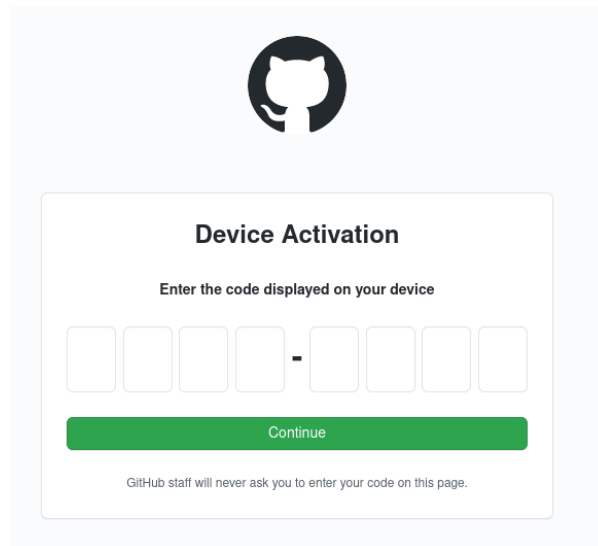
**Figure 6.1:** verification\_uri site of Google

### 6.2 GitHub

GitHub uses a device\_code of 40 characters with a hexadecimal base. Although the resulting entropy of the device\_code is not as high as the one used by Google, it is sufficiently high to fulfill the criterion mentioned in Section 5.1.

The `user_code` is implemented with eight case-insensitive alphabetic characters. Information about limiting `user_code` entry attempts is not available.

GitHub also alerts the user on `user_code` entry that the staff will never ask the user to enter his code. The warning serves prevention measure against phishing attacks as described in Section 4.5 and increases the user's awareness that the process can be utilized for malicious purposes.



**Figure 6.2:** `verification_uri` site of GitHub

### 6.3 Amazon

Amazon offers multiple services that use the Device Grant. For example, a user's Amazon account can be connected to a smart TV to use Amazon Prime or other devices such as smart speakers that come with Alexa [AMAZON].

On the website reached by `verification_uri`, Amazon shows an example `user_code` of XB5GQ. Assuming 36 characters and a length of 5, the entropy would be  $36^5 \approx 2^41$ . Amazon does not show the user which device he is currently registering in the verification process and only prompts the input of the `user_code` and authentication. In contrast to Google, Amazon does not show the privileges granted to a client during authentication [GOOGLE].

## 7 Conclusion

The current RFC specifications leave the Device Grant vulnerable to various attacks. While some of them can be easily prevented if all recommended specifications are carefully implemented, others are not sufficiently covered in the current version of the RFC. Many choices are left to the developers at the IdP, some of whom might not be aware of the consequences. As we have shown in a practical setting, an implementation of the minimal requirements entails a number of flaws that can rather easily be exploited by an attacker. Not all resource providers implement best practices.

In particular, we suggest that the specification should demand a minimum entropy of the `device_code` and prohibit the display of the `device_code` both on the client device and in the authentication process (see 4.2). Instead, the authentication process should show the name or type of device that is registered and may even include a location. Furthermore, the RFC should prohibit displaying advertisements on the login page that is reached by the `verification_uri_complete` since otherwise the owner of the advertisement can easily obtain the `user_code` (see 4.1.2). In Section 4.1.2, we illustrated the effects of rate-limiting. We strongly recommend any AS to employ some kind of rate-limiting functionalities to make a brute-force attack less feasible, especially for `user_code` entry. In addition, in the current version of the RFC the `user_code` is not required to be unique which is not security-relevant but may result in an error if the AS lacks proper exception case handling. We suggest to recommend the `user_code` to be unique.

The greatest weakness of the Device Grant is the user. Hence, great efforts should be made to make the process as transparent as possible to the user and make him aware of what he is doing. Additional steps, such as informing a user via email of a new device connected to his account, should be contemplated. We consider it best practice to explicitly ask the user if he wishes to connect the device with his account after entering the `user_code`. Showing the name of a device in this step significantly reduces the risk of a phishing attack since the user can recognize suspicious devices. Privilege escalation is another indicator of an ongoing attack, hence, privileges granted to the client should be shown in the login process.

In conclusion, we have seen that the above listed points have a major impact on the security of the Device Grant. Therefore, they fulfill the RFC criterion of “MUST” rather than “SHOULD” and ought to be included as security-relevant aspects of the protocol. The objective of the RFC specifications is to formalize and establish a secure protocol standard, which is why the current version requires revision.

# Bibliography

- [AMAZON] *LWA for TVs and Other Devices*. Accessed: 2021-03-15. URL: <https://developer.amazon.com/docs/login-with-amazon/other-platforms-cbl-docs.html> (cit. on p. 26).
- [FKS16] D. Fett, R. Küsters, G. Schmitz. “A Comprehensive Formal Security Analysis of OAuth 2.0”. In: *CoRR* abs/1601.01229 (2016). arXiv: 1601.01229. URL: <http://arxiv.org/abs/1601.01229> (cit. on p. 4).
- [GOOGLE] *OAuth 2.0 for TV and Limited-Input Device Applications*. Accessed: 2021-03-24. URL: <https://developers.google.com/identity/protocols/oauth2/limited-input-device> (cit. on pp. 25, 26).
- [KKK20] J. Katic, M. Kotowsky, D. Krüger. “Informelle Sicherheitsanalyse des OAuth Device Flow”. In: (May 2020) (cit. on pp. 4, 6, 16).
- [RFC211997] S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. BCP 14. <http://www.rfc-editor.org/rfc/rfc2119.txt>. RFC Editor, Mar. 1997. URL: <http://www.rfc-editor.org/rfc/rfc2119.txt> (cit. on p. 24).
- [RFC224699] T. Dierks, C. Allen. *The TLS Protocol Version 1.0*. RFC 2246. <http://www.rfc-editor.org/rfc/rfc2246.txt>. RFC Editor, Jan. 1999. URL: <http://www.rfc-editor.org/rfc/rfc2246.txt> (cit. on p. 16).
- [RFC674912] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. <http://www.rfc-editor.org/rfc/rfc6749.txt>. RFC Editor, Oct. 2012. URL: <http://www.rfc-editor.org/rfc/rfc6749.txt> (cit. on pp. 16, 24).
- [RFC862819] W. Denniss, J. Bradley, M. Jones, H. Tschofenig. *OAuth 2.0 Device Authorization Grant*. RFC 8628. <https://tools.ietf.org/html/rfc8628>. RFC Editor, Aug. 2019. URL: <https://tools.ietf.org/html/rfc8628> (cit. on pp. 13, 16, 19, 23, 24).