Front-End Restriction Evasion on YouTube for Enhanced User Experience (UX)

*YouTube Ad Skipper*

**Introduction**

YouTube is one of the world’s largest video platforms, generating revenue primarily through advertisements and its YouTube Premium subscription service. YouTube Premium offers an ad-free experience, exclusive content, and offline playback, providing an alternative revenue stream that reduces reliance on ad-based monetization. The success of this model depends on users opting to pay for an uninterrupted experience instead of bypassing ads through third-party tools.

YouTube’s ad system is structured to balance revenue generation with user engagement. The platform delivers three primary ad formats:

* **Skippable Video Ads** – Viewers can skip these after 5 seconds.
* **Non-Skippable Video Ads** – Must be watched in full (15-30 seconds).
* **Bumper Ads** – Short, 6-second, non-skippable ads.

This ecosystem supports advertisers, content creators, and YouTube itself, ensuring free content remains available while generating revenue. However, automation tools that bypass ad restrictions threaten this model.

**Threats to YouTube's Revenue Model**

The presence of tools or techniques that bypass YouTube’s ad structure poses a significant threat to both YouTube’s business model and its ecosystem. If ads can be skipped or circumvented entirely, YouTube could face:

1. **Revenue Loss**
2. **Impact on Creators**
3. **Reduced YouTube Premium Growth**
4. **Exploitation of System Weaknesses**

**Detectability by Ad Blocker Detection Software**

One critical question surrounding such ad-skipping mechanisms is whether they are detectable by software that identifies ad blockers. To answer this, we need to compare the functionality of traditional ad blockers with automation tools like the one explored in this project:

**How Traditional Ad Blockers Work**

1. Ad blockers typically prevent the browser from downloading ad-related content by intercepting network requests to ad servers and many ad blockers remove or hide ad-related DOM elements using predefined rules.
2. **Detection by YouTube**: Ad blockers can often be detected because they disrupt the normal flow of ad requests, leave traces in network activity, or interfere with specific JavaScript-based ad delivery.

**How This Tool Differs**

Unlike traditional ad blockers, this tool:

**Does Not Intercept Ad Requests**:

* It does not block network requests or interfere with ad delivery. Ads are fully loaded and played as intended, ensuring no server-side disruptions and because ads are downloaded and initiated, YouTube’s servers will still register the ad as served.

**Mimics User Actions**:

* The tool simulates user interactions, such as clicking the skip button or fast-forwarding the video. These actions are indistinguishable from real user behaviour, making them hard to detect. Since it works purely on the client-side, it does not trigger any anomalies in YouTube's server-side tracking.

**Relies on JavaScript Injection**:

* Instead of hiding ad elements, it modifies the behaviour of existing DOM elements (e.g., using ***currentTime*** to skip to the end of an ad or ***playbackRate*** to accelerate ad playback). These changes are temporary and localized to the client-side, leaving no detectable footprints.

**Why It Is Hard to Detect**

* Normal Ad Loading
* No External Signatures
* User Simulation
* Localized Manipulation

**Why YouTube’s Ad Structure Matters**

YouTube’s ad system sustains the platform’s free content model while ensuring advertisers reach their audiences. Skipping ads via automation weakens this structure, reducing revenue and engagement metrics. Understanding these bypass techniques provides insight into potential countermeasures and highlights the need for stronger front-end ad enforcement.

**Vulnerabilities**

A blue and white background

Description automatically generated with medium confidence**VULN 1**

As shown in the above image, during advertisements playback speed cannot be controlled by clients, this option however is available during regular playback.

A screen shot of a computer

Description automatically generated

This exploit utilizes the playback function and applies it to advertisements to reduce watch times for ads with the option of increasing viewing speeds to 20x the normal speed. For this however we need to identify when an ad is being played.

A screenshot of a computer program

Description automatically generated

***.ytp-ad-progress-list*** indicates whether an ad is playing or not allowing us to differentiate between videos and ads, we can use this as a trigger for the ***execute\_script()***, this is done by identifying if the CSS class exists, if the condition is met then the script function can be run. Furthermore, it not only indicates whether an ad is playing but also when it ends by then disappearing which indicates to the code the condition is no longer True and will then return the playback speed to normal.

A screen shot of a computer

Description automatically generated  
Despite YouTube’s UI buttons being separate from the ***<video>*** element, it is responsible for playing videos and ads as this media form is loaded within the element. This makes modifications like ***playbackRate***, ***currentTime***, etc. possible.

The **src** attribute is responsible for loading the video from YouTube’s servers, however the file cannot be directly accessed via the browser to ensure the raw video file cannot be altered. This segmented streaming model indicates that the most viable option of approach is by targeting the dynamic nature of YouTube’s algorithm with JavaScript injection being the main focus as videos are not presented as fixed files.

|  |  |
| --- | --- |
| YouTube’s Restriction | Bypass Method |
| ***src*** is hidden and raw video file cannot be altered. | Instead of altering the raw video file we can change playback properties using JavaScript Injection |
| YouTube dynamically loads video segments | Instead of manipulating video streams, changes can be forced via ***.html5-main-video*** |
| YouTube disables playback speed control during ads | JavaScript can be injected into the browser to override ***playbackRate*** during ads |
| Ads are controlled via JavaScript, not by embedding them in the video element | By detecting the element ***.ytp-ad-progress-list*** only appeared when an ad is running we can identify ads and automate skipping. |

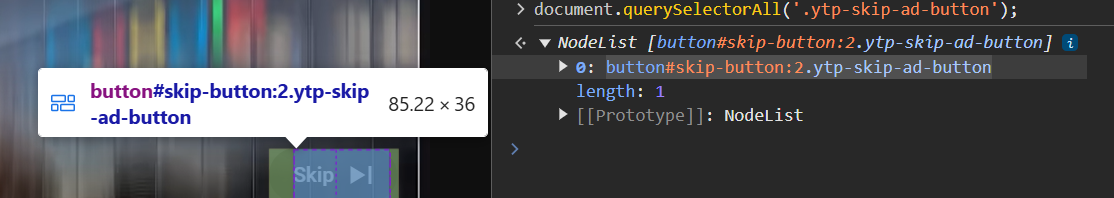
A computer screen shot of a program code

Description automatically generated  
This segment of code, particularly the highlighted section increases the playback speed to 20x the normal speed, making a 30 second un-skippable ad 1.5 seconds long.

**VULN 2**

Despite the reduced speed of ads, it is not the best approach. A redundant optimization layer is required for this code to enhance its effectiveness. The skip ad function serves as a fallback mechanism that activates when accelerated playback exceeds a predefined threshold, in this case 5 seconds, ensuring seamless ad removal with minimal delay. The reasoning behind this implementation is that despite un-skippable ads not being allowed to exceed 15 seconds on mobile and 30 seconds on TV according to YouTube’s official guidelines, skippable ads do not have any restrictions on length which is problematic for this code as a 5 minute/600 second ad is 15 seconds when sped up to 20x. This means an extra 14.75\* seconds are spent watching an ad when the skip button has been made available 5 seconds in or 0.25 seconds sped up. Due to logistical issues such as CPU usage the code refreshes a check every 2 seconds to ensure too much memory isn’t being occupied by the code this still leaves an A screen shot of a computer

Description automatically generatedA screen shot of a computer screen

Description automatically generatedunnecessary amount of time spent on an ad.

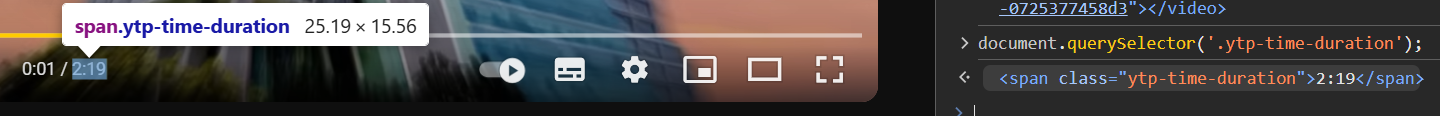
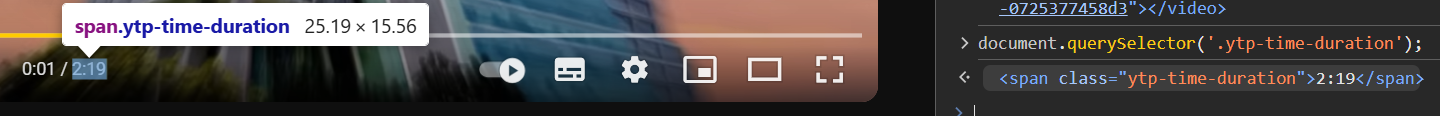
The solution is an IF-ELSE statement with the parent IF condition checking whether an ad is playing while the nested IF statement checks if ***.ytp-ad-skip-button*** exists indicating a skippable ad may now be skipped.

The ad detection system leverages YouTube’s Document Object Model (DOM) to determine the best way to bypass advertisements.

|  |  |  |
| --- | --- | --- |
| **Detection Parameter** | **Execution Mechanism** | **Action Taken** |
| Ad Detected (ad-showing) | Check for skip button (***.ytp-ad-skip-button***) | Click Immediately if available |
| Skip button not found | Check if ad is non-skippable | Increase playback speed to 20x |
| Ad progress monitored | Monitor ad duration and skip availability | Fallback to skip if duration increases threshold |

**VULN 3**

Another restriction in YouTube is the ability to strictly only be able to move the progress slider during video playback and not ads as it would then become possible to scroll to the end of the ad without watching it. ***currentTime*** can be exploited to bypass this, by identifying the length of the video using the ***.ytp-ad-progress-list*** and ***.ytp-time-duration*** in conjunction with one another as a way of identifying an ad is running and the duration of the ad to alter the **currentTime** allowing the client to jump to sections in the ad. A way of implementing this might be identifying the text within the ***.ytp-time-duration*** CSS class, converting it to seconds and altering the **currentTime** to be just before the video ends so that the ad is registered as viewed rather than there being potential for an error.



Using the value derived from ***.ytp-time-duration*** we can convert the time into seconds, e.g. 2:19 = 139 seconds. Then using selenium, we can pass the JavaScript placeholder ***arguments[0]*** and subtract 0.5 from the 139 seconds which moves the ***currentTime*** from the initial value to 138.5 seconds from the start.

A computer screen shot of a code

Description automatically generated

***YouTube Ad Skipper* (V2) Explanation**

The V2 version of the code improves upon the previous implementation by introducing enhanced functionality and handling additional edge cases encountered during ad detection and bypass. Specifically, this version includes:

Interacting with the .***ytp-ad-skip-button-modern*** element:

* This element appears on ads that are either watched to completion or, in the case of this code, fast-forwarded to the end using the ***currentTime*** property.
* The code now detects and interacts with the .***ytp-ad-skip-button-modern*** element to ensure that ads are fully dismissed, even after being fast-forwarded.
* By integrating this functionality, the script avoids unnecessary delays caused by lingering ads that still display the advertiser's logo or website information after completion.

The V2 version also integrates the following ad-bypassing methods into a single streamlined system:

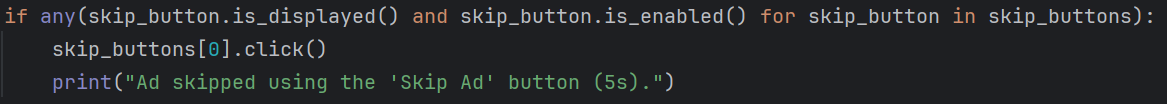
* **Fast forwarding non-skippable ads**

A screen shot of a computer

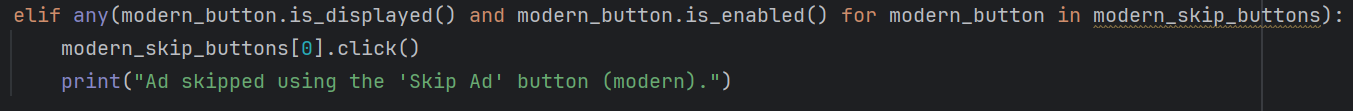
Description automatically generatedFor ads that do not have the option to skip, the script calculates ad duration and jumps directly to the last second using the ***currentTime*** property this eliminating most of the ad playback time while still allowing the video to progress naturally ensuring it cannot be found by ad blocker detection software.

* **Skipping Ads via Buttons**

The script checks for the presence of skippable ad buttons, including:

Skip Button shown during ad (5s later): ***.ytp-skip-ad-button.***



******Skip Button shown after ad: ***.ytp-ad-skip-button-modern.***

******

Both work using a search of all instances of their respective ***.ytp*** elements in the DOM field using ***find\_elements()***. It will then check if any of these buttons are visible (***is\_displayed()***) and is enabled (***is\_enabled()***). If both requirements are True, the button is clicked ensuring the dismissal of ads.

* **Enhanced handling of edge cases**

After fast-forwarding non-skippable ads to the last second, the script rechecks for the .***ytp-ad-skip-button-modern*** element and clicks it if it becomes available to ensure that ads are fully dismissed, even in scenarios where the button appears after the ad has technically ended.

**Skills Demonstrated:**

**DOM (Document Object Model) Manipulation & Ad Detection Mechanisms**

* I explored how YouTube structures ads in its DOM, identifying elements such as ***ad-showing***, ***ytp-skip-ad-button, ytp-ad-progress-list*** to reliably detect when an ad is playing.
* I learned how to track dynamic changes in DOM elements, particularly for detecting transitions between ads and regular video playback.

**Application:**

* Built a detection logic that differentiated skippable ads from non-skippable ones using DOM-based cues.
* Used conditions to adapt the system’s behaviour based on the type of ad detected.

**Optimising Efficiency in Automation**

* Designed a multi-layered decision-making framework to prioritise efficiency:
  + Primary Action: Skip ads when the Skip button is available.
  + Secondary Action: Use ***currentTime*** manipulation for non-skippable ads to jump to the last second.
  + Fallback: Accelerate playback speed using ***playbackRate*** if neither skipping nor ***currentTime*** manipulation is possible.

**Application**:

* Optimised ad-bypass process to limit waiting time without impacting normal video playback behaviour.

**Debugging and Problem Solving**

* Used Chrome DevTools such as element and console to inspect and troubleshoot dynamic web elements.
* Debugged issues like false positives in ad detection and unintended playback resets during testing.
* Refined the system iteratively to handle edge cases (e.g., mid-roll ads, UI inconsistencies).

**Application:**

* Built resilience into the script by handling exceptions and improving detection accuracy in real-world conditions.

**Web Automation Exploitation using Selenium.**

* Gained hands-on experience with Selenium WebDriver to automate browser interactions and learned to identify, locate, and manipulate web elements dynamically using techniques like ***find\_element()***, ***DOM queries*** and ***execute\_script()***.

**Application:**

* Automated tasks such as clicking the Skip button, detecting when ads start and stop, and manipulating video playback.

**Client-Side Bypass (Front-End Restriction Evasion)**

* Understood the concept of bypassing front-end UI restrictions by interacting with hidden or dynamically loaded elements.
* Explored how YouTube relies on client-side playback logic for ad completion and user interaction.

**Application:**

* Bypassed ad enforcement mechanisms to create a seamless user experience while understanding the limitations of client-side security.

**JavaScript Injection**

* Injected JavaScript code into YouTube’s DOM using Selenium’s ***execute\_script()*** method to manipulate critical attributes of the HTML5 video player, such as ***currentTime*** and ***playbackRate***.

**Application:**

* Automated processes like skipping to the last second of an ad (currentTime) and increasing playback speed for non-skippable ads.

**User Experience (UX) Restriction Bypass**

* Understood how UX elements like the Skip button and playback controls are designed to enforce ad viewing.

**Application:**

* Automated user actions like clicking the Skip button and interacting with UI elements, mimicking natural user behaviour.

**Proof of Concept (PoC)**

**Summary**

This proof of concept demonstrates a method to bypass YouTube's advertisement system through automation and front-end manipulation. The vulnerability exploits YouTube's reliance on client-side ad rendering and playback logic, allowing:

1. Automatic skipping of skippable ads via UI interaction.
2. Forced completion of non-skippable ads using ***currentTime*** manipulation in the HTML5 video player.

|  |
| --- |
| **Google’s High-Severity Abuse Risk Examples** |
| Disrupting revenue models (Ad Fraud/Ad Evasion) |
| Automation that bypasses engagement metrics |
| Economic impact (e.g., bypassing paid features [YouTube premium]) |

The bypass reduces YouTube’s ability to deliver ads as the code is easily implementable on all desktop devices and if fine-tuned for distribution purposes may have the effect of greater loss in ad revenue and may fall under Google’s guidelines of abuse:

**Impact**

The bypass reduces YouTube's ability to deliver ads, impacting its:

* **Ad revenue model:** Advertisers are charged per view or impression, both of which are bypassed undermining advertiser confidence.
* **User engagement tracking:** Prevents accurate reporting on ad view metrics.

**Steps to Reproduce**

Tool: Selenium WebDriver

Browser: Google Chrome (valid user profile)

Target URL: <https://www.youtube.com>

**Script Execution**

[Code Provided in ***x*** directory]

**Exploit Details**

**Skippable Ads**

* **Observed Behaviour**: Skippable ads show a Skip button after 5 seconds.
* **Exploit**: The script identifies the Skip button using .***ytp-skip-ad-button*** and programmatically clicks it as soon as it becomes available.

**Non-Skippable and Skippable Ads**:

* **Observed Behaviour**: Non-skippable ads must play to completion, typically lasting 15–30 seconds, whereas skippable ads may only be skipped after 5 seconds of playback.
* **Exploit**: The script retrieves the ad's total duration using ***.duration*** and sets the ***currentTime*** property to the last second, forcing ad completion.

**Fallback Mechanism**:

* If the script detects issues with ***currentTime***, it defaults to speeding up playback ***playbackRate = 16***.

**Reproducibility**

**Ethical Considerations**

This PoC is provided for **educational purposes only** to demonstrate how front-end restrictions can be bypassed using automation. It does not leverage server-side vulnerabilities or unauthorized access.

**Proposed Fixes**

Validate playback server-side:

* Ensure that ad progress and playback metrics are validated on the server rather than relying on client-side properties like ***currentTime*** or ***playbackRate***.

Disable programmatic interaction with Ad UI:

* Prevent automation tools like Selenium from interacting with ad-related DOM elements like ***.ytp-skip-ad-button.***

Obfuscate Video Properties

* Mask or encrypt properties like ***duration*** or ***currentTime*** to prevent direct manipulation.

**Impact** **Assessment**

If exploited at scale, this vulnerability could:

* Reduce YouTube’s ad revenue by up to 100% for affected users.
* Undermine advertiser confidence in YouTube's ad platform.
* Create a loophole for mass abuse by automation tools or browser extensions.