Narrowbeer: A Practical Replay Attack Against the Widevine DRM

Florian Roudot, Mohamed Sabt

Narrowbeer: A Practical Replay Attack Against the Widevine DRM

Florian Roudot
Univ Rennes, CNRS, IRISA

Mohamed Sabt
Univ Rennes, CNRS, IRISA

Abstract

Streaming services like Netflix, Prime Video, and HBO Max rely on DRM solutions to ward off piracy. By enabling the distribution of encrypted content, DRM systems prevent subscribed users from downloading the streamed content, as well as unauthorized users from having access to it.

Google Widevine, one of the most deployed DRMs, provides a fully software-based solution on desktop platforms to ensure portability. In this paper, we empirically investigate the security protections implemented by Widevine to counter an attacker tampering with its interactions within its environment, namely with the operating system and the hosting browser. Focusing on randomness and time, we uncover new flaws in the Widevine license acquisition process, particularly targeting the freshness and expiration of the licenses. To demonstrate the effectiveness of our findings, we develop Narrowbeer, a practical replay attack allowing legitimate users to generate never-expiring licenses, and enabling unauthorized users to reuse these licenses to access premium content without subscription. Finally, we validate our attack against real-world streaming services by succeeding in repeatedly playing the same license on different desktop devices.

1 Introduction

Context and Motivations. The number of streaming services users has increased steadily over the past few years, as an estimated 165.9 million people had at least one subscription in 2024 in the United States (US) [19]. The digital distribution of media via the Web presents significant challenges for streaming platforms, with piracy being the foremost issue. A recent study indicates that more than 80% of digital piracy originates from streaming [4]. This piracy undermines the business model of most streaming services, which rely on monthly subscriptions, requiring customers to pay a regular fee to access a catalog of premium video content. As a result, content providers leverage Digital Rights Management (DRM) technology to protect media from piracy.

Modern DRMs deliver content in an encrypted format. The user's computer must decrypt the content before it can be rendered on their device screen, which is where the Content Decryption Module (CDM) comes into play. The CDM is the DRM component on the client side and can be implemented using software (e.g., in desktop environments), secure hardware (e.g., Intel SGX for Blu-ray [21]), or a combination of both (e.g., ARM TrustZone in smartphones). Obviously, the CDM requires the decryption key in order to display the encrypted content. The method by which the key is delivered to the CDM varies, but typically involves a protocol between the CDM and the content provider. During this "license acquisition" process, the content provider determines whether it trusts the CDM and whether the user has permission to access the content. If the licensing authority, henceforth called the license server, approves the request, it issues a "license" containing the relevant key material to the CDM, alongside its usage rights, such as the expiration time.

There was a time when streaming services needed to inconveniently adapt their websites for each DRM using third-party plugins because of the diversity of existing DRM solutions that are not even interoperable. Striving to improve the viewing experience of watching movies on the Web, the World Wide Web Consortium (W3C) published Encrypted Media Extensions (EME) as a W3C Recommendation or Web standard [22]. Roughly, the EME protocol abstracts the license acquisition process and related actions, allowing content providers to deploy their streaming services everywhere on the Web regardless of the DRM system being used. An unexpected outcome of the advent of EME is the wide adoption of a limited number of DRM solutions owned by OS and platform providers: Google Widevine, Microsoft PlayReady, and Apple FairPlay. Widevine is distinguished in being both cross-platform (Windows, Android, Linux, and macOS) and cross-browser (Firefox, Edge, Chrome, and all other Chromium-based browsers). This portability comes with a price: Widevine deploys a software-only CDM module running as a user-land process in desktops, deemed less secure than their hardware-based or kernel-embedded counterparts.

This compromise between portability and security has pushed Widevine to integrate a plethora of proprietary software security mechanisms to counter powerful attackers having full control of their devices. In addition, Widevine has put effort while designing its EME protocol, which was formally proved to prevent eavesdroppers from intercepting the keys as they are transmitted over the Web [6].

Existing work [9, 10, 15, 20] mostly attempt to recover the media in unencrypted form by frontally attacking the CDM software protections. Thus, a spiral of patch-and-hack has been continuing for years, implicitly assuming that the Widevine DRM is secure only if its software protections remain unbroken. Moreover, the scope of these works is limited, since only the effectiveness of different software protections were highlighted without providing any particular insight into Widevine within its DRM ecosystem. Unfortunately, despite its wide deployment and adoption, little has been done to examine Widevine with other attack vectors and different attacker models. This paper shifts the focus from the commonly studied protection, which is the confidentiality of license keys, to other security properties of the Widevine CDM. In particular, we study enforcing expiration times and preventing the replay of media licenses against an attacker who maliciously intercepts calls to OS resources.

Contributions. In this work, we present three main contributions. **First**, we analyze the integration of Widevine into EME desktop browsers, highlighting the interactions between Widevine and the EME ecosystem. This analysis aims to identify key interactions between Widevine and its hosting environment to access important resources for security, especially time and randomness. Two views are presented: the browser view (native calls between the browser and Widevine) and the system view (calls to OS resources). **Second**, we examine the potential actions of an unprivileged user against the Widevine DRM without considering breaking its implemented software protection mechanisms. Thus, we rather focus on the security implications of a system attacker intercepting and manipulating critical system calls used by Widevine to obtain time and random values. Accordingly, we present our study, which answers three related research questions supported by extensive empirical evidence and observations. During our investigation, we identified a design flaw corresponding to the security goal of properly enforcing the expiration time of licenses. Indeed, an attacker controlling the time can use a license beyond its specified expiration time. This issue arises because Widevine CDM does not impose that time cannot move backward during the Widevine protocol workflow. **Third**, we introduce a new vulnerability called Narrowbeer, allowing an attacker to replay a valid license forever. With Narrowbeer, anyone, including users without subscriptions, can use old licenses to watch the related content on any desktop device. We rely on the EME protocol to build a next-generation torrent client that is both fast and lightweight, where pirates share small licenses, only a few kilobytes in size, and exploit weaknesses in Widevine to access media content while using the content providers' infrastructure for streaming. This approach offers a significant advantage for Narrowbeer over previous methods, as it eliminates the need to store large volumes of pirated content on servers controlled by attackers.

These findings highlight the inherent limitations of relying solely on software-based protections, such as cryptographic mechanisms and anti-debugging techniques, to secure DRM systems. As a result, our research calls for a more comprehensive approach to DRM security that incorporates both software and hardware-based protection mechanisms, particularly in environments where attackers may have access to system-level resources. The main insight of our research is that contrary to popular belief, Widevine's weakest link lies in its random number generation, not in its white-box cryptography or obfuscation techniques.

2 Background

2.1 Digital Rights Management

Digital Rights Management (DRM) systems protect copyrighted content from piracy. While different DRM solutions exist to protect different kinds of content on various platforms, this paper focuses solely on DRMs for streamed audio/video content over the Web. At first, these DRMs were included within the media plugins, such as Adobe Flash or Microsoft Silverlight, which were necessary to play media content on web browsers. With the advent of the `<video>` tag in HTML5, the need for such media plugins disappeared. Consequently, DRM systems became standalone and were then delivered as their own plugin. The inner workings of a DRM system can be summarized through three main components:

CDN: The Content Delivery Network encrypts the copyrighted contents through some content packager and makes them available for streaming.

License Server: It delivers the required license (i.e., decryption key and associated usage rights) to play the protected media upon request by a user and the verification of its legitimacy (e.g., valid subscription).

CDM: The Content Decryption Module is in charge of loading and protecting the obtained license on the user's device to decrypt and decode the streamed content while enforcing its usage rights.

Here is a simplified workflow for a DRM whenever the user selects some content to display on their favorite streaming website. First, the CDN delivers a protected version of the media. Once the content is retrieved, the CDM generates a license request destined for the license server, which replies with a license response containing the decryption key(s) and

associated usage rights. The client returns this response to the CDM, which loads the content key(s) in memory. Finally, the user's device can play the protected content via the CDM.

2.2 W3C EME

The diversity of proprietary DRM solutions has led the W3C to publish the Encrypted Media Extension (EME) to standardize the interface between webpages and the CDMs through a web browser. The EME standard, whose first draft was edited by Google, Microsoft, and Netflix, defines a JavaScript API and a unified workflow for acquiring the required licenses while streaming protected content on the Web. The advent of EME was much awaited in the streaming industry: all major browsers (e.g., Chrome, Edge, Safari, and Firefox) supported it before its standardization in 2017. The main reason behind this wide adoption is that streaming websites can finally work seamlessly across various browsers and DRM systems without third-party plugins. An overview of the EME protocol is detailed in Appendix A.

2.3 Google Widevine

Widevine is a closed-source proprietary DRM system owned by Google. Its portability on a wide range of systems, including Android devices, smart TVs, game consoles, as well as Windows, Linux, and MacOS desktop browsers, makes it one of the most adopted DRM solutions, as it is used by major streaming platforms such as Netflix, Prime Video and HBO Max. Its early deployment into the Chromium project has allowed this DRM to be integrated into most browsers as their default EME CDM. In this paper, we focus on the desktop implementations of Widevine leveraged by Chrome and Firefox on both Linux and Windows. This instance of Widevine comes as a shared library dynamically loaded and sandboxed by the browser when a web page needs it.

3 Insights into the Desktop Widevine EME

As presented in section 2, Widevine is heavily used by web-based content providers through the EME standard interface. Related works reverse-engineer the internals of the Widevine CDM in Android environments [15,16] and desktop ones [10], as well as the content of the opaque messages in EME for both license acquisition and renewal [6,17]. In this section, we thoroughly dissect the integration of Widevine into EME-supporting desktop browsers. In particular, we detail all the interactions of the Widevine implementation with the EME ecosystem. This will help us to better frame our empirical security investigation in subsequent sections by pointing out the relevant interactions to tamper with in order to assess the security of Widevine. Our goal is not to explore all the semantics and technical details of each interaction but to depict a correspondence between the EME Javascript API specified in [22] and the Widevine CDM API as a native shared library loaded by the browser. Below, we provide two views of the desktop Widevine EME: the browser view (native calls between the browser and the CDM) and the system view (calls to OS resources).

3.1 The Browser View

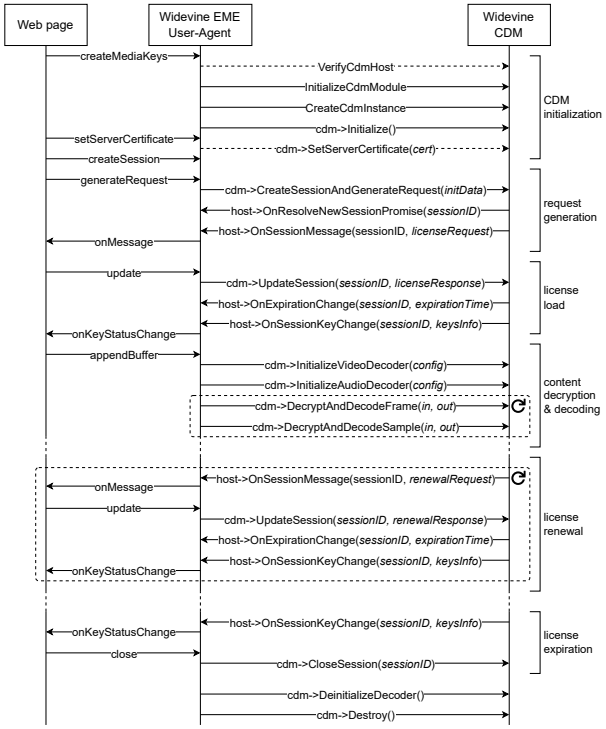

Figure 1: The Native Browser Workflow of Widevine EME

As indicated previously, EME was designed so that web pages of content providers can work on browsers without any third-party plugin, regardless of the underlying DRM system. Indeed, the burden of interacting with the supported DRM system on the user's device is now shifted to EME user-agents, namely browsers. In other words, each browser implementing EME needs to instantiate the EME JavaScript API for each supported CDM. In this paper, we focus on Widevine, the Google DRM solution used by both Firefox and Chrome (along with all other Chromium-based browsers, such as Edge). In Figure 1, we summarize the functions and workflow of Widevine on a desktop (Windows or Linux) browser. Our findings were obtained by investigating the Firefox support of Widevine, which is very similar to the Chrom(e)(ium) one, through a two-pronged approach. First, we leverage the EME Logger extension [8] to log all the EME JavaScript method calls and events. Second, we replace the Widevine CDM library with our CDM acting as a wrapper to Widevine in order to log every native call destined for the

Widevine CDM. Below, we highlight the important steps.

Initialization. The Widevine CDM comes as a library dynamically loaded by the EME user-agent, the browser in our case. Once loaded in memory, the user-agent can ask the CDM to verify its integrity with the `VerifyCdmHost` function. Part of the Verified Media Path (VMP) feature of Widevine, this function verifies the signature of binaries (on Firefox Windows: firefox.exe, plugin-container.exe, widevinecdm.dll, xull.dll). These signatures will also be added to the license request so the license server can verify them. Then, the CDM gets initialized, and the user-agent retrieves an object to interact with the CDM using `CreateCdmInstance`. After initializing this instance of the CDM, the user-agent can proactively provide the CDM with a certificate from the license server. In [17], authors found that this certificate is used for the privacy mode.

License Generation. Upon the CDM initialization, the user-agent can ask to generate a new license request with `CreateSessionAndGenerateRequest` called with content-specific initialization data. This function will create a new session associated with a random 16-byte session ID and an empty key wallet. The next step is to generate the license request. Studied in [6, 15], the request contains, among others, a random 16-byte request ID, the content key ID(s), a request time, and two 4-byte nonces. It also includes a client ID containing the CDM certificate and some user-agent information. Widevine offers a *Privacy Mode* to encrypt the client ID using AES-CBC. Thus, a random privacy key and IV are generated and added to the license request. The privacy key gets RSA-OAEP encrypted using the license server certificate. The CDM signs the license request and returns it to the user-agent, which forwards it to the license server.

License Load. Using a Session Key, the license server derives three new keys from the license request: the Asset Key, the MAC Server Key, and the MAC Client Key. The Asset Key is used to encrypt the Content Key(s) in the license response. The original timestamp and nonce are included, along with the license policies specifying how the CDM can use the content and for how long. Based on the signatures of the binaries, the license server also determines the VMP status (`PLATFORM_VERIFIED`, `PLATFORM_TAMPERED`, or `PLATFORM_UNVERIFIED`) and reports it in the license response. Leveraging different cryptographic keys, the confidentiality of the Session Key, as well as the integrity of the license response are guaranteed by the Widevine protocol. The license server sends the license back to the browser, which transfers it to the CDM via the `UpdateSession` function. Then, the CDM starts by deriving the same three keys as the license server using the Session Key and the request buffer. The nonce is verified, and the expiration time of the Content Key(s) is computed. Finally, the Content Key(s) are decrypted and loaded into the CDM memory, making it ready to decrypt the protected content.

License Use and Renewal. The browser is now ready to request the decryption of content using the appropriate functions (`DecryptAndDecodeFrame` for video and `DecryptAndDecodeSample` for audio). The protected content is decrypted, decoded, and returned to the browser to be played. Depending on the policies, the license can be renewed before its expiration. Once a policy-specified delay has passed, the CDM generates a renewal request containing the initial request ID and the initial request time. In addition, the time of the renewal request is included, along with a counter. If mandated by the policies, the client ID can also be included and protected by the privacy mode if activated. A MAC tag is computed over the entire request before sending it to the license server. A renewal response is returned to the CDM to update the corresponding policies specifying the new license duration. Upon integrity verification by the CDM, the expiration time of the license is modified according to the new policies. If no (successful) renewal occurs, the license eventually expires, and the opened session is closed after notification of expiration. Finally, the created CDM instance is destroyed, for instance, when the associated web page is closed.

3.2 The System View

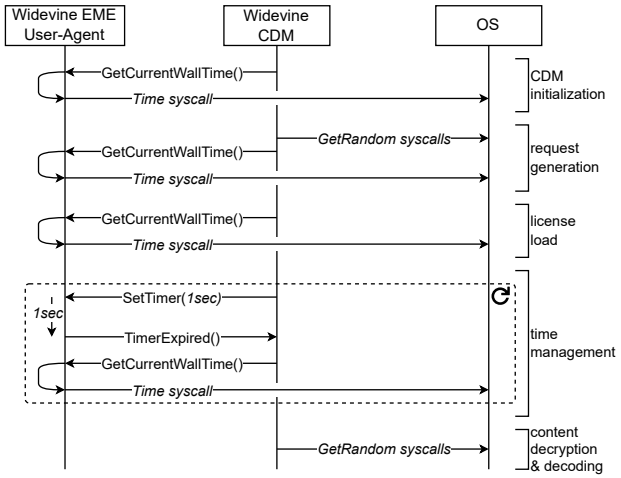

Figure 2: The System View of Widevine EME as Requiring Time and Randomness.

In order to enforce its security properties, Widevine requires generating random bytes and accessing the host time. Indeed, as indicated in [6], time is required to manage license expiration, and randomness is required to mainly protect licenses. The portable design of Widevine has mandated that its CDM would rely on the underlying operating system for these critical resources. Here, we describe their use during a typical workflow of the Widevine protocol. To this end, we leverage the `ptrace` utility on Linux to inspect the relevant system calls performed by Widevine. As for Windows, we check that the corresponding system calls are actually issued

in an equivalent manner. The observed workflow is summarized in Figure 2. No difference was observed between the Linux and the Windows implementations.

Starting from its initialization, Widevine regularly needs to retrieve the host time. Peculiarly, we notice that Widevine does not directly ask the operating system to get the current time. Instead, it goes through the browser (its EME user-agent) via a callback mechanism set during initialization. Thus, it is the browser that performs the system calls to obtain the needed timestamp for Widevine.

During the request generation, Widevine repeatedly fetches random bytes from the system. The random bytes are mainly used to enforce the Privacy Mode (both CBC-AES and RSA-OAEP encryption) and avoid replay attacks (4-byte nonce). Here, it is Widevine that issues the needed calls to the operating system. A fresh timestamp is also fetched and added to the request. When loading the license, once again, Widevine asks for the time to check the validity of the license in regard to their policies and the current time. We do not detail the system calls for renewal as they are similar to the ones for the license acquisition process, except for the nonce.

From this point on, Widevine relies on a system of timers in concordance with the browser: several timers are set in a loop. When a timer expires, the browser calls the Widevine method `TimerExpired` to wake it up. In response, the CDM asks the browser for the time via the method `GetCurrentWallTime` and executes further routines depending on the returned value. A new one-second timer is created every second to maintain an internal clock for Widevine, thereby allowing it to keep track of time for checking license key validity. Other longer timers are also set to keep track of license usage. System calls for randomness are also needed during the decryption and decoding of the protected content. However, we were not able to map these bytes to any of the enforced security properties.

4 Problem Statement

4.1 Threat Model

DRM attacks, also known as Man-At-The-End (MATE) attacks, describe a strong model in which a malicious actor has (physical) access to the victim device [1]. DRM attackers are assumed to tamper with the hardware or software of a target system without restriction. Generally speaking, DRM systems have to consider "all-mighty" attackers that have control over the system on which the media contents to protect are played. This is because an attacker can choose the device on which they display the contents that they want to pirate.

In this paper, our threat model is distinct from the general DRM attacker model, in which the attacker is assumed to have complete control over the hardware and software they run. Our attacker is simpler: they can just install and run a new binary on their own device. They can also run some scripts without any particular privilege. In addition, we exclude all software-based attacks based on CDM vulnerabilities. This is because software-only DRM systems mostly implement defenses, such as code obfuscation and deployment of anti-debug techniques [2, 10, 15], that are not expected to resist against powerful attacks indefinitely, but rather to slow or deter an attacker. In addition, such vulnerabilities are more related to software protection that can be easily patched than to the inherent design of DRMs. Moreover, based on [6], we assume secure the Widevine implementation of the EME protocol, including the leveraged cryptographic mechanisms. Our assumptions limit the scope of our attacker actions, but they make any related vulnerability more practical.

To sum up, we investigate what an unprivileged user can do against the Widevine DRM on a secure, fully updated running environment, including the Widevine CDM, the operating system, and the browser. In other words, we would like to question the following statement: the security of a DRM system holds if and only if the software security of its CDM implementation remains unbroken. In subsequent sections, we will show flaws in the Widevine DRM without challenging any of the applied software protection mechanisms.

4.2 Security Requirements

In [11], authors introduce the concept of a perfect DRM system that protects against piracy. As shown by Delaune *et al.* in [6], piracy can be hard to model because it involves security notions beyond the confidentiality and integrity of messages. To overcome this, they suggest seven security goals to better formalize the naive definition of piracy, which is "preventing DRM-protected media from being distributed outside the DRM ecosystem". The literature has mainly investigated confidentiality (Goal 1), which results in obtaining a digital copy of the protected content outside the CDM. In this paper, we rather focus on Goal 3 (freshness of licenses) and Goal 4 (enforcing expiration time).

Of particular interest, we study these two properties: (1) a given valid license obtained from a license server can be loaded at most once on its corresponding CDM, and (2) a license cannot be used beyond the expiration time specified by its license server. Thus, in our security model, the attacker's goal is to target the running CDM software to break the aforementioned properties, undermining the security of Widevine as a DRM system. Indeed, licenses contain the necessary decryption keys to display protected content. If they got loaded more than once, then the associated content can also be played more than once without authorization from the license server or the content provider. Most importantly, the illegal distribution becomes wild if the content licenses are not bound to a particular device. This is because an attacker can generate licenses from a valid account on some device and then distribute them for other attackers to load on their own devices. Furthermore, for the second property, we assume the license server expects the enforced expiration time by the CDM to

be relatively defined according to the request time (not the load time) embedded in the received license request. For instance, a license would expire on January 2nd 2025, if the request time was January 1st 2025 and the policies indicate 24 hours of license lifetime. Therefore, the license server expects that the license cannot be used even for a single minute when it is loaded on January 3rd. In this paper, we investigate whether the Widevine CDM holds the expectations of the license server related to our threat model.

4.3 Research Questions

Despite the wide deployment of streaming services relying on DRM systems to protect their contents, little has been done to understand how the most deployed DRM on Desktop browsers, namely Widevine, resists in practice against intercepting calls to OS resources. Our paper takes the first step in this direction and explores the security implications of a system attacker controlling the outputs of critical system calls issued by the Widevine CDM. To better frame the scope of our study, we also consider Widevine implementations supporting the Verified Media Path (VMP) feature, which allows Widevine to verify the integrity of the running environment. We aim to gain insights into the following question: **What are the security implications of tampering with the CDM system calls for acquiring time and random?** By addressing this question, we introduce new attack vectors that show how ineffective software-only protection mechanisms can be when the whole DRM workflow is considered. Accordingly, we present our study that answers the following questions with ample empirical evidence and observations:

- **RQ1.** How would the Widevine CDM react when tampering with its system communication and application programming interfaces?
- **RQ2.** How would the license acquisition process be impacted when all random numbers get fixed?
- **RQ3.** How would the effective license expiration time be altered when the CDM system of timers got controlled?

5 Empirical Investigation

5.1 RQ1: CDM Interactions

The Widevine CDM, as a software module, interacts with two external entities: the OS via system calls and the EME user-agent through the CDM public API and a set of callbacks. The RQ1 explores the possibility of tampering with these interactions. In particular, we study whether the CDM implements any related security defense for the license acquisition process within a browser environment. By tampering, we mean intercepting targeted calls to arbitrarily modify their arguments or return values.

Methodology. To investigate Widevine, we first created a standalone C++ program on Linux to dynamically load and interact with the Widevine CDM shared library, without the overhead of a browser. Using a host environment similar to the one employed by Firefox, we were able to complete a simple license acquisition following the EME workflow.

Next, to observe browser interactions with Widevine, we built a shared library that wraps the original Widevine CDM. This library reimplements each public function and method from the Widevine API by logging the call and its arguments before forwarding them to the original library. We then replaced the original Widevine shared library with ours, allowing us to retrieve logs of every call made by Firefox to Widevine and vice versa. Then, several real content providers were visited to analyze their EME workflow.

To go further, we leverage `ptrace` on Linux to hook the CDM system calls. At first, our hook only observes the different arguments in read-only mode. Then, we attempt to complete the EME workflow despite modifying the return values. Finally, we reiterate our experiments on Windows by creating a similar wrapper to observe the calls and their arguments. Here, we first performed static analysis to identify system API calls made by Widevine on Windows. Then, the MinHook library [13] was used to intercept these calls by injecting trampolines to hook the targeted functions. Indeed, MinHook works by overwriting the prologue of the hooked function with a `JMP` instruction pointing to a *detour* function. It also allows us to call the original function through a *trampoline* function, which is a clone of the function prologue followed by a `JMP` instruction redirecting to the rest of the original function.

As the DRM 'black box' model raised security concerns amongst the Web community, the browser process managing the CDM gets loaded inside a GMP (Gecko Media Plugins) sandbox. For our study, we disable the sandbox protection, as it restricts `ptrace` capabilities on Linux and prevents the process from loading additional libraries.

Results. Defined in the methodology, our wrapper loads the Widevine CDM in its memory, and directly calls its API instead of the browser. Upon a complete EME workflow, our observations depend on the underlying operating system or, to be more precise, on the VMP feature. Indeed, we did not notice any difference on Linux, and all calls were successful. However, on Windows, the `VerifyCdmHost` function detects that the calling module is not the browser user-agent. The CDM does not stop. However, the license server is warned of this tampering through the license request. Our analyses detect three different behaviors: (1) the protected content is played, implying that the license server did not consider the VMP status, (2) the content provider issues a warning mes-

sage that the quality of the content has been degraded, and (3) the content provider refuses to reply with a valid license response. The difference between Linux and Windows is that VMP is not supported on Linux, and therefore the VMP status is always `PLATFORM_UNVERIFIED`.

Regarding hooking system calls, we check whether we can still play protected contents, while intercepting several calls and modifying their arguments or return values. The experiments were performed on both Linux and Windows. In both cases, the CDM did not crash, and the content was successfully played, which means that the Widevine CDM does not implement any protection against hooking these calls despite their importance. Moreover, VMP does not detect that the calling user-agent is being traced.

Key Takeaways. Recall that the Widevine CDM depends on the hosting browser for its system of timers (refer to Figure 2). Therefore, our wrapper can easily tamper with the Widevine timers on Linux machines, since we could control all calls to `TimerExpired()`. However, we did not adopt this approach because it might have excluded several content providers on Windows clients from our subsequent observations and defined attacks. Henceforth, we rely solely on hooks to intercept calls for time and randomness, which we explore the consequences in the following research questions.

5.2 RQ2: CDM Randomness

Randomness is critical for any security protocol. We assume that the Widevine EME is no exception. Therefore, we expect to observe weaknesses resulting from replacing all random values with fixed ones. In this research question, we mainly focus on license acquisition to assess whether its security might be broken when suppressing randomness. By security, we mean the confidentiality, integrity and anti-replay security goals as defined by Delaune *et al.* [6].

Methodology. Based on our previously defined hook, we intercept all the `getrandom` system calls used by Widevine to retrieve random values. We then analyze each of these system calls, and modify the content of their argument buffers with fixed values. A complete EME workflow is attempted, and all exchanged messages are logged and analyzed. As for Windows, while analyzing the Widevine CDM on Windows, we found the function `SystemFunction036` (documented as `RtlGenRandom` by Microsoft) from the `Advapi32.dll` Windows API. Likewise `getrandom` on Linux, this function fills buffers with random values used in the license request. Thus, we leverage the MinHook library to intercept the `RtlGenRandom` function similarly to Linux.

Results. During a license acquisition, the Widevine CDM issues several calls to get random values. We first map each of these calls to the corresponding fields in the resulting license request. When the privacy mode is not enabled, we correctly modify the following fields: the 16-byte session ID, the 16-byte request ID, the 4-byte nonces, and the 82-byte RSA-PSS signature salt. Two requests having all these fields fixed for the same media (in addition to the request timestamp) will have the same resulting RSA signature over the whole request. We continue our mapping when the privacy mode is enabled and successfully control the following fields: the 16-byte AES privacy key and the 16-byte initialization vector (IV). Both values are used to CBC-encrypt the client ID. We confirmed our findings by decrypting the parsed client ID with the fixed values. We notice that the ciphertext of the encrypted privacy key is never equal between different requests despite our hooks fixing all random values. Thus, we suspect the RSA-OAEP seed is being generated without issuing a system call.

Then, we continue our investigation by empirically determining which of these values are part of the derivation buffer used to derive the Asset Key, the MAC Server Key, and the MAC Client Key. To do this, we generate a license request with all the random values fixed, and retrieve the corresponding license response from the server. We then start a new session and generate a new request with all the random values fixed except one. Recall that the primary condition for the CDM to load a license response is to obtain the right derived session keys. Therefore, if the previous license response can be loaded, we exclude the value that was not fixed from the derivation buffer. Our experiments indicate that, in addition to the request timestamp, only the request ID and the first nonce are part of the derivation buffer when the privacy mode is not enabled. Following the same methodology, we were not able to successfully load a previous license response into a new session even when all random values were fixed. Consequently, we conclude that the ciphertext of the encrypted privacy key that we did not succeed in fixing is included in the derivation buffer. Moreover, both the privacy key and the CBC IV are also included, since, as shown by Patat *et al.* [15], the (encrypted) client ID is part of the derivation buffer.

Key Takeaways. The important random values to fix are the 16-byte request ID and the first 4-byte nonce. With the privacy mode, we should additionally fix the 16-byte AES key, the 16-byte IV and the 128-byte RSA-OAEP seed. Unfortunately, our attempts to fix the seed were not successful. We address this challenge in the next section while describing our attack.

5.3 RQ3: Effective Expiration Time

The business model of many content providers relies on the idea that users cannot consume media beyond their subscriptions. For instance, a subscription should be for at least 10 days if the user takes 10 days to watch their favorite shows. Strictly speaking, the subscription is only needed to acquire the required licenses. Therefore, a naive attack against DRM systems would be to end one's subscription as soon as a

browser tab was opened and loaded the license for each show to watch. However, this attack is not possible in practice because licenses have a (relatively short) expiration time defined by their license servers and enforced by the CDM. In this research question, we explore the possibility of breaking this protection when tampering with the Widevine times.

Methodology. Time syscalls are optimized via vDSO by being loaded into userspace on Linux. This makes `ptrace` unable to detect them. To address this, we disable the vDSO mechanism by iterating through the process stack during its creation and erasing the vDSO base address in the auxiliary vectors. We then hook the `gettimeofday` system calls used by the Widevine host with `ptrace` to modify the returned time. Regarding Windows, we use MinHook to hook `GetSystemTimeAsFileTime` and `timeGetTime`, which the CDM host uses to fetch the time for Widevine.

Results. We define several time values used during the license acquisition process. T_1 and T_2 represent the timestamps at which the license request is generated and the license response is loaded, respectively. The TTL (Time-to-Live) of the license is the value defined by the license server and specified in the license response. T_3 is the timestamp at which the license will expire: $T_3 = T_1 + \text{TTL}$. We define $eTTL$, the effective TTL, as the time period during which the license is still usable; the license expires when $eTTL = 0$ as it decreases every second after the load, or more precisely after the call to the `TimerExpired()` function. We can easily see that $eTTL = T_3 - T_2 = \text{TTL} - (T_2 - T_1)$. Naturally, we have $\text{TTL} \geq eTTL$, since we should always have $T_2 \geq T_1$.

Our first investigation research line is to study the impact of modifying T_1. We move T_1 back and forward and trigger a license acquisition on some content providers. Regardless of its value, we notice that the license server always responds with a valid license starting on T_1.

Then, we investigate the impact of altering the time flow on the $eTTL$ by modifying T_2, the timestamp at which the license response is loaded. Our experiments distinguish three possible behaviors. First, after the load, we continue to fix the time such that $t_i = T_2$ for all i in time. Based on our definitions, no call to `TimerExpired()` would occur. Thus, the $eTTL$ would never decrease, implying that the license would never expire. Nevertheless, this does not happen in practice. Indeed, fixing the time for the Widevine CDM also fixes the time for the video buffering, causing the video to stop loading after a short period. Second, we only modify T_2 at load time, then move back or forward to the current time t. Of course, we still have that $t > T_1$, since t represents the current time just after load. We observed that the outcome of only modifying T_2 depends on how T_2 is moved on the timeline. If T_2 was moved forward (i.e., $T_2 > t$), then there is no impact on the $eTTL$. However, if T_2 was moved back (i.e., $T_2 < t$), then the $eTTL$ is decreased by $t - T_2$ on the first `TimerExpired()`, then by 1 every second. As a matter of fact, both cases are very similar since $eTTL = TTL - (t - T_1)$ always holds, where t is the current time when the first timer expires. The last case is when we control the current time relatively to the modified T_2, so that $t_i = T_2 + i$, where i represents the seconds. Here, the $eTTL$ is decreased as usual: by 1 per second.

Key Takeaways. Our findings lead to an attack regarding the second security requirement defined in subsection 4.2. Indeed, an attacker controlling the time can use a license beyond the expiration time specified by the license server. The vulnerability is caused by the Widevine CDM not enforcing that $T_2 \geq T_1$ must hold. The attack can be achieved by two different methods. First, we can increase the lifetime of a license by modifying the timestamp at which it is loaded such that $T_1 > T_2$, and then letting the time progress relatively to T_2. For example, with a generated license request on $T_1 = 01/01/2025$ at 00:00, we load the license response with a TTL of 24 hours, modifying the timestamp to $T_2 = 01/01/1970$ at 00:00. Even though the license should expire on $T_3 = T_1 + \text{TTL} = 01/02/2025$ at 00:00, this does not happen, providing that $eTTL = \text{TTL} - (T_2 - T_1) = 55$ years and 24 hours. Second, the same attack is achieved if we fix the timestamp T_1 in the future and then return to the present at load time. Our illustrative example goes as follows: if we generate a license request with $T_1 = 01/01/2125$ at 00:00 and load a 24-hour license on $T_2 = 01/01/2025$ at 00:00, the loaded license will expire on $T_3 = 01/02/2125$ at 00:00, and its effective TTL will be $eTTL = T_3 - T_2 = 100$ years and 24 hours.

6 The Narrowbeer Attack

The Widevine CDM is used to protect premium media assets on the Web. The most studied protection is the confidentiality of the license keys so that no component, except the CDM, can decrypt the media and distribute it for free. In this paper, we focus on another security property of the Widevine CDM: replaying media licenses. Indeed, Widevine generates fresh values for each session to load any license only once. Therefore, you always need your Netflix subscription, for instance, to watch any media, even if you have already consumed it. This section shows that this property can be easily bypassed on desktops: a valid license can be replayed forever.

6.1 Intuition

Based on our observations regarding the raised research questions, we identify a potential vulnerability in the desktop implementation of Widevine, allowing an attacker to reuse a previously acquired license response. Such a replay attack would allow an attacker to bypass the license server, which completely undermines the security model of DRM systems. Indeed, the protected content could be played repeatedly regardless of its associated usage rights and expiration time.

In addition, the license could be shared with other attackers who could also play the protected content on their desktop machine, thereby bypassing any security enforced by the concerned content providers, including verification of clients' subscriptions via authentication.

The primary condition for the CDM to load a license response is to obtain the right Asset Key, MAC Server Key, and MAC Client Key. Consequently, a replay attack succeeds whenever the Session Key and the derivation buffer used to derive these three keys remain constant between two sessions. The Session Key has no reason to change, as it is already stored in the license response. However, it becomes more complicated for the derivation buffer since it contains some fresh random data and a timestamp related to the request time. The CDM also appends a random nonce to the license request, which is also included in the response.

Our findings indicate that a license response generated within a session can be loaded into another session if the two derivation buffers of the two sessions match; otherwise, the CDM would not be able to derive the right MAC Keys to verify the response integrity, nor the right Asset Key to decrypt the Content Key(s). This requirement also involves the nonce that must match between the request and the response inside the same session. Our empirical experiments show that a straightforward implementation of such an attack does not succeed when the Privacy Mode is enabled, which is the case for Windows for all content providers and on Linux for some content providers, especially Netflix. This is because if the privacy mode is enabled, the client ID is encrypted by a privacy key using AES-CBC. This key is then encrypted using RSA-OAEP with the public key of a certificate provided by the content provider. In this case, the derivation buffer also contains the IV for the CBC mode and the RSA-OAEP encryption of the AES privacy key. OAEP is a probabilistic encryption scheme that requires fresh random data during encryption, namely the seed. The problem, at least for a DRM attacker, is that the Widevine CDM on an x86-64 desktop environment does not issue any system call to obtain the needed random buffer. Below, we tackle the aforementioned challenge and fully succeed in the attack on premium content providers.

6.2 Technical Implementation

6.2.1 Settings

We implement our attack on Linux, specifically Debian 12 and Windows 11 machines. We use the Firefox web browser (version esr-128) as the EME User-Agent with the Widevine CDM version 4.10.2710.0. Thanks to our responsible disclosure, this Widevine CDM version has since been deprecated by Google and replaced by a new version that adds more obfuscation to the randomization process. We did not implement the attack on MacOS systems since, as shown by [17], the Widevine CDM implementations of MacOS and Windows behave similarly as two operating systems supporting VMP. To sum up, this attack works on the latest version before the patch of the Widevine CDM on a modern browser in a generic way between Linux and Windows without breaking any obfuscation or enforced memory integrity protection.

6.2.2 Time

Figure 2 indicates that time is used during the whole process of license acquisition, namely in both generation and load. As for license generation, the license request contains a timestamp indicating the request time. This timestamp is part of the derivation buffer, implying that we must fix it for a given value. Fortunately for us, we can pick any value for the timestamp (past, present or future), because the timestamp embedded in the license request is not verified by the license server when issuing the license response. Moreover, we must also control the timestamp for the license load, since it controls the resulted expiration time of the license (refer to subsection 5.3). Ideally, the timestamp for the load should be smaller than the one for the request in order to extend the expiration time enforced by Widevine.

Recall that the Widevine CDM relies on the browser to keep track of time via `GetCurrentWallTime`. In its current version, this method uses the `gettimeofday` syscall on Linux, and a combination of the functions `GetSystemAsFileTime` and `timeGetTime` on Windows. The Widevine VMP prevents us from directly modifying the browser source code as its executable is signed by Widevine. Thus, we hook the necessary functions and syscall to modify their return value, following the methodology detailed in subsection 5.3. In our implementation, the simple approach of indefinitely fixing the time results in the content stopping after a few seconds. This is because Firefox leverages the same time function to manage the content buffering. To prevent that, we selectively fix the time during license generation: we fix the time to a fixed value T until the salt of the license request signature has been obtained via a `getrandom` or `RtlGenRandom` call, easily identified by the size of its argument buffer: 82. Then, we continue to update the time returned by the hooked functions to $T + i$, where i is the number of seconds that has passed since the generation of the license request.

6.2.3 Randomness

The Widevine CDM relies on the execution environment to generate random values. Mostly, it leverages the underlying operating system through a mechanism of system calls. Again, the simple approach of disabling all randomness of Firefox or Widevine by forcing fixed values can cause the Widevine process to crash. Therefore, we only apply our hook, described in subsection 5.2, on the Widevine sandboxed process depending on the size of the argument buffer. This process allows us to fix the fresh values embedded in a newly generated license

request, such as the 16-byte request ID and the 4-byte nonce, the 16-byte IV and the 16-byte AES privacy key. Nevertheless, this is not enough when the privacy mode is enabled because the random seed used to RSA-OAEP encrypt the privacy key cannot be overwritten by our hook.

Indeed, Widevine performs this encryption using the statically linked BoringSSL library. While analyzing the code source of BoringSSL, we notice that the generation of random values is done using the Intel RDRAND instructions on compatible systems. If unavailable, BoringSSL, and therefore Widevine, resorts to the underlying operating system through system calls. We look to overcome this limitation because we want our attack to apply regardless of the underlying execution environment. At first, we emulate the RDRAND instructions with Intel's Pin utility [12]. This solution worked nicely on Linux but caused Widevine to crash on Windows, probably because of some software protection mechanisms that are out of the scope of our paper. Then, we modify Widevine, so that the RDRAND branch is never taken. Now, the RSA-OAEP seed can also be overwritten using our hook. Technically, we could not directly modify the binary because of VMP. Instead, we alter the Widevine image in memory by dynamically modifying two bytes at runtime. Unlike the Widevine core, the statically-linked BoringSSL is not protected against such a dynamic modification of its instructions. Somehow, this weakens our threat model in which we assume that the Widevine CDM cannot be dynamically attacked. However, it makes our attack as portable as Widevine by working similarly on Linux and Windows, excluding some specific OS details.

6.3 Attack Overview

In Narrowbeer, we define two types of attackers: the Harvester to collect valid licenses, and the Consumers to replay these licenses. Both the Harvester and the Consumers use our hooking methodology to control time and randomness while generating license requests. To do so, they require either the Ptrace capability on Linux or the SeDebugPrivilege on Windows; two privileges that an attacker with full control over their device can easily grant. We distinguish between these attackers since only the Harvester needs a valid subscription. Below, we describe the three steps of the attack. The role of each attacker is depicted in Figure 3.

First Step: Collect (Encrypted) Media Tracks. The streaming protocol DASH relies on a file called manifest containing metadata for each media forming the content to stream (different quality video files, different language audio files). In particular, the manifest includes the protection scheme and the URLs of the streamed content. In the case of DRM-protected content, these URLs, which anyone can access, lead to the encrypted form of the media. When watching content on a streaming platform, the web page retrieves this file to generate the license request. For the attack, the Harvester collects the URL where the manifest is stored to share it later with the Consumers.

Second Step: Harvest License Responses. The most straightforward approach is to hook EME calls on Javascript and get the license responses. This approach has a caveat: the licenses do not contain all information related to the derivation buffer, especially when the privacy mode is enabled. Optionally, we fix the request timestamp to '9999999999L', which corresponds to the year 2286, to have a license that "never" expires when loaded in memory. In addition, in order to simplify the exploitation part, all random values included in the derivation buffer are fixed, which is not strictly necessary because some of them can be found when parsing the licenses to be replayed. The Harvester sends this 'rogue' but valid request to the license server. Finally, the resulting license response is intercepted and stored by the Harvester (a subscriber user) to share it with the Consumers (non-subscriber users). The next step is to trick Widevine to replay these license responses.

Third Step: Replay License Responses for Free. The last step is to use Shaka Player [7] with the collected mpd file (from step 1) to play the media. Under the hood, Shaka Player calls the Widevine CDM to establish a new EME session. Similar to step 2, we intercept and control a set of system calls performed by the Widevine CDM during the generation of the license request. Finally, instead of contacting the license server, we just replay the previously harvested license response (from step 2). The replay attack succeeds because the derivation buffers in steps 2 and 3 are identical, and therefore all verification, including the different cryptographic operations, during load would be carried off without failure.

6.4 Evaluation

Summarized in Figure 3, we apply the previously described techniques to build a complete proof of concept, allowing us to play protected content leveraging Narrowbeer on Widevine. Narrowbeer consists of an executable to launch and attach to Firefox, including its child processes, either by using Ptrace on Linux or injecting our hooking library on Windows. Once the Widevine process is found, Narrowbeer fixes the time and random values to obtain the required derivation buffer. Then, we create a web page with a modified version of Shaka Player, an open-source video player developed by Google [7]. Shaka Player is a JavaScript library that supports playing DRM-protected content via EME. Our version of Shaka Player does not strictly follow the EME workflow: instead of sending the license request to the license server to obtain a fresh license, it looks into a database for a previously retrieved license and forwards it to the CDM.

We validate our attack on real-world Widevine license servers. First, to avoid legal action, we test the attack on some protected content available on the Shaka Player demo website.

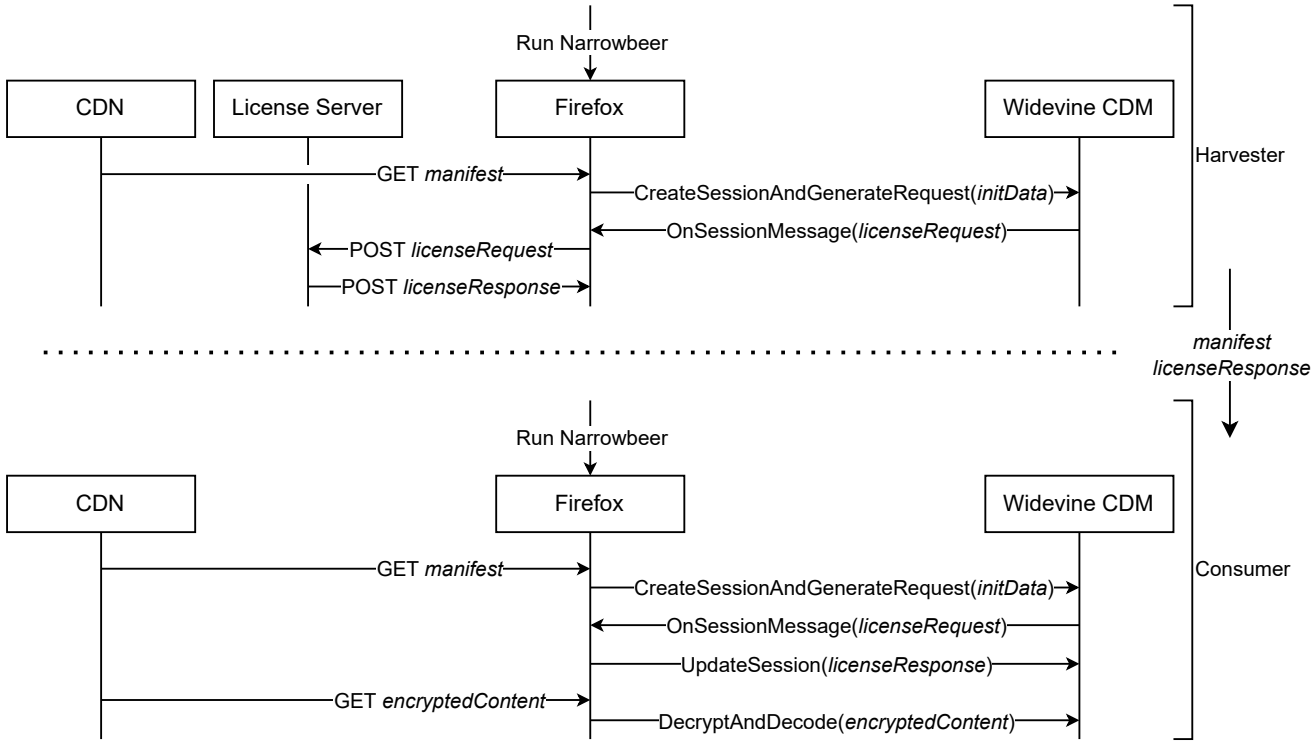

Figure 3: Replay Attack Workflow

We start by running Narrowbeer while playing the content on the original website. Here, we do not need to watch all the content; we just retrieve the URL of the content manifest and use the EME logger web extension to grasp the license response. These data are stored in a JSON file of only a few kilobytes and will be downloaded by the Consumers. Later, on another desktop, we run Narrowbeer again and browse to the modified Shaka Player web page that takes the aforementioned JSON file as input. Shaka Player uses the manifest URL to retrieve the content metadata and start the license acquisition process. While Widevine generates a new license request, Narrowbeer once again runs in the background and fixes all the necessary values. The load succeeds, and the protected content is displayed without contacting the license server on the Consumers' devices.

Second, we went further and evaluated the effectiveness of Narrowbeer against two premium services with a clear disclosure policy: Netflix and Prime Video. The implemented steps are similar to the ones described above, except for collecting the URLs of the contents manifest, which depends on each content provider. The attack succeeds, allowing us to watch both Netflix and Prime Video without a subscription.

7 Discussion

7.1 Towards Torrent 2.0

The application scope of Narrowbeer is quite wide. This is because, unlike in Android, there is no individualization of the Widevine CDM on PC, which means that all computers share the same Widevine Device Key. Therefore, the last step of consuming the stolen licenses is independent of the step of collecting them. In particular, this implies that a user with a subscription can collect license responses and share them online with anyone unless the root key of the CDM has been updated, which does not occur regularly. Generally speaking, this is useless because a license response cannot be reused. However, with our attack, anyone, including users without subscriptions, can use old license responses to watch the related media. This opens the door for a next-generation torrent client (both lightweight and fast), where people share quite small license responses (a few kilobytes), and exploit Widevine weaknesses to watch the media while leveraging the content providers infrastructure for the streaming part. This represents a great advantage of Narrowbeer compared to previous attacks since there is no need to store pirated content on some servers owned by the attackers. Indeed, we leverage the EME protocol for our own benefit, where Narrowbeer follows the workflow in Figure 4 except for the communica-

tion with the license server. In a nutshell, Narrowbeer, as the next-generation torrent, works as follows:

1. Pirates (harvesters with subscription): get licenses that expire in the year 2286 and distribute them.
2. Consumers (without a subscription):
    - Get the Narrowbeer binary and execute it.
    - For each media, torrent the associated data (JSON file). That is pretty fast since it is only a few KBs.
    - The protected content can be played on the browser, including with its original subtitles.

Again, we highlight that attackers do not need to get cumbersome with terabytes of videos. They can just distribute the Narrowbeer binary with a few KBs per media.

7.2 The Attacker Capabilities

In Linux, attackers (Harvesters and Consumers) need the 'ptrace' capability. Root access is not required, since the tracing process (Narrowbeer) directly creates the traced process (Firefox). In Windows, we rely on the 'OpenProcess' Windows API, which requires the 'SeDebugPrivilege' to obtain full access to the targeted process. This privilege, default to the Administrator group, can be granted for a given user.

While it is possible that ptrace or SeDebugPrivilege could be disabled, we did not consider this as a significant limitation, as an attacker with full control over their device (root or admin access) can permanently grant the required privileges.

7.3 The Sub-HD Limitation

The main limitation of Narrowbeer is that only sub-HD movies can be played because premium content providers, such as Netflix and Prime Video, restrict the video quality for Widevine desktop users. However, anyone satisfied with sub-HD quality can use this method to watch premium content without subscribing to multiple content providers. For those seeking higher quality, video upscaling solutions are rapidly improving. On Windows, Firefox now includes the NVIDIA RTX Video Super Resolution feature [14], which allows any user with an NVIDIA RTX GPU to upscale videos to a higher resolution. We tested the attack on a compatible machine and observed an apparent enhancement in the video quality.

7.4 Responsible Disclosure

All experiments and proof of concept attacks were performed on resources under our full control. The Narrowbeer web page was hosted locally. We limited our study to Widevine test platform and streaming services with clear responsible disclosure policies (e.g., Netflix and Prime Video). On these platforms, we limited the number of pirated resources. Following our responsible disclosure process, our findings have been reported in a timely manner to all concerned parties: Google (Widevine) as the distributor of the vulnerable binary, and Netflix and Prime Video as Widevine's users. We first notified Widevine, since the vulnerability affects their own binary. Being directly impacted, we also warned the streaming services just in case Widevine was reluctant to notify them. We suggested a 90-day disclosure period. All parties acknowledged the attacks, confirming that their systems are vulnerable and need patching. Moreover, considering the high impact of the attack, all parties have rewarded us via their bug bounty program; however, they act differently in the patching process. Netflix initially paid no attention to the issue, claiming that it was not their vulnerability to fix. Then, our disclosure report suddenly caught their interest, and they helped us communicate with Widevine to get a patch as soon as possible. Prime Video was quite responsive and tried to address the vulnerability by implementing some time enforcement on their license server. Their patch does not allow the generation of license responses when the request time exceeds three days. They asked us to evaluate their patch, and we showed them how their patch does not protect against our replay attack, especially since once the Harvester acquires the license response, the license server is no longer contacted. The Consumers, who can modify the time at will, can still replay the license.

As for Widevine, the communication was not easy. First, we contacted Widevine using their own process[1]. We got no answer until the Netflix security team started a discussion thread with them. Afterward, the Netflix team advised us to report the vulnerability through Google's bug bounty program, which is the owner of Widevine. The Google team announced that the vulnerability would be patched but committed to no timeline. While we were not kept informed by the Google/Widevine team about the patch development, we were monitoring Firefox Nightly for a new Widevine version. We were not even informed when the new patched version (specifically, the CDM version 4.10.2830.0) was released in August 2024, nor when the vulnerable Widevine CDM was deprecated three months later. With further investigation, we find that the patch breaks the replay attack by using an additional internal source of randomness. However, being out of the loop of the patching process, we were not asked to evaluate the effectiveness of the implemented patch which we are convinced could be easily broken once the internals of this source are identified and controlled. Furthermore, the vulnerability of having never-expiring licenses still remains exploitable in the current version. One last note about this responsible disclosure: while we got publicly acknowledged in the Hall of Fame of Netflix and Prime Video, no public statement was issued by Google or Widevine. This is unfortunate because it condemns the DRM ecosystem to the

[1] https://www.widevine.com/contact

constantly failing security-by-obscurity principle. Our goal is to improve the knowledge about DRM and not provide copyright infringement tools. We believe that DRMs should quit the realm of secrecy in order to embrace better security that is well-assessed by the community.

7.5 Proposed Countermeasures

Since the attacker has full control over the machine on which the Widevine CDM runs, very little can mitigate this attack. Indeed, anti-ptrace mechanisms or increasing the complexity of the randomness generation process may only extend the time an attacker needs to reverse-engineer the CDM. However, it will not address the inherent problem of the underlying issue. Therefore, stronger defenses are needed. One might suggest implementing Widevine as a kernel driver in order to prevent the hooking of its libraries from user-land processes. However, we question the effectiveness of such a mitigation, since this might only shift the attacks to the kernel space, which would still be valid regarding the threat model of DRM systems. In addition, technologies depending on this protection, such as anti-cheat solutions on Windows, are repeatedly broken [5]. Another mitigation might be for Widevine to implement its own random generator without relying on the well-known system interface to get randomness (e.g., the getrandom system call on Linux). This solution removes the risk of system calls being intercepted but remains vulnerable to reverse engineering despite obfuscation. Specialized CPU instructions like Intel RDRAND can also be notably harder to intercept, however, they limit the portability unless combined with a default portable solution (e.g., the BoringSSL approach). The challenge would be to get enough entropy for the Widevine cryptographic operations whenever a software-only solution is considered. So, what about hardware solutions?

Hardware-backed CDMs, like Widevine L1 [15], mitigate this attack by generating the license request and its random values within a Trusted Execution Environment (TEE), ensuring that sensitive operations are isolated from the rest of the system and protected from potential compromises. The problem is that hardware CDMs on desktop machines are not available yet. We do not expect this current state of affairs to improve, since Intel SGX has been deprecated and is no longer deployed on newer CPUs. A promising opportunity for Widevine might be to push its L1 CDM to the newest computers on ARM chips supporting ARM TrustZone. In a nutshell, our attack shows how brittle a generic software-only CDM implementation can be. Our suggested mitigations tend to leverage either hardware specific extensions (e.g., RDRAND instructions on Intel, or ARM TrustZone), or OS specific features, such as those offered by Windows for anti-cheat solutions. This is not ideal for a portable CDM like Widevine, as this would restrict the number of potential clients of a given content provider using Widevine. In other words, Widevine values portability and would, therefore, avoid relying on RDRAND instructions or hardware modules like TPMs, even though they constitute a viable solution to secure the randomness generation process.

8 Related Work

Although DRMs are a commonly used technology nowadays, there is relatively little literature exploring their internal mechanisms. Indeed, breaking a DRM system is still illegal in many countries. Exemptions were only added to the Digital Millennium Copyright Act (DMCA) in the United States in 2018 to allow security researchers to work on DRMs.

Such exemptions have encouraged a new line of public research. Most related contributions focus on techniques to extract the CDM root keys so that an attacker can obtain the license keys and decrypt the protected content. In [10], Tomer Hadad details his reverse-engineering process to extract the root-of-trust of the desktop implementation of Widevine. Patat *et al.* reveal the cryptographic mechanisms used by Widevine on Android, and manage to recover the L3 root-of-trust. Qi Zhao [24] shows that even the Widevine implementation running inside a Trusted Execution Environment (TEE) in Android smartphones can be broken. More recently, Adam Gowdiak [9] claims to have extracted the private ECC keys from the PlayReady CDM.

Another common attack target is to remove the protection from the encrypted content. In [3], David Buchanan develops a new generic attack on DRMs leveraging the MPEG-CENC file format to decrypt protected content. The cause of the attack is twofold: (1) the absence of authenticated encryption in CENC, and (2) a feature of the x264 and x265 video codecs that allow forging custom MP4 files bypassing the encoding process. Prior to this work, Ruoyu Wang *et al.* [23] broke the PlayReady CDM by semi-automatically retrieving the decrypted but not yet decoded content from the process memory.

Recently, new attack vectors were explored. In [16], Patat *et al.* expose to what extent content providers do not follow the security guidelines of Widevine in their streaming services. They also discover a privacy leak impacting the end-user caused by the browsers' misimplementation of Widevine EME [17]. In [6], Delaune *et al.* study the Widevine EME protocol by reverse-engineering the content of its opaque messages. They define several security goals for the EME enforcement layer and formally verify Widevine security against these goals. Their formalization led them to discover a new vulnerability in Widevine allowing attackers to arbitrarily modify a license usage right for their benefit.

9 Conclusion

In this paper, we have explored the security vulnerabilities of the Widevine DRM system under a weakened DRM threat

model: an unprivileged attacker who can install binaries and run scripts on their own device. We focused on the practical aspects of tampering with the Widevine CDM, particularly its system calls related to time and randomness, to assess the robustness of Widevine against different attacks.

Through our investigation, we presented an empirical exploration of how the Widevine CDM reacts when its interactions via APIs and system calls are tampered with. Our findings demonstrate how DRM security can be broken by fixing random numbers and controlling time. Finally, we developed Narrowbeer; a practical replay attack in which an attacker can obtain a never-expiring and reusable license, enabling unsubscribed users to access protected content.

A key research question remains: to what extent other DRM systems, such as PlayReady of Microsoft, are vulnerable to our attacks? An interesting future work would be to systematically analyze different software-based CDMs and the impact of tampering with their process of acquiring time and randomness. Such a study might either reveal a generic attack vector against a wide set of DRMs, or uncover some underexplored security mechanisms effectively protecting their CDM.

10 Acknowledgments

This work received funding from the French National Research Agency (ANR) under the project DRAMA (grant agreement No. ANR-22-CE39-0005), and the France 2030 program managed by the ANR under the project SVP (grant agreement No. ANR-22-PECY-0006).

11 Open Science

In accordance with Usenix Open Science policy, we provide all the source materials necessary to reproduce the Narrowbeer attack in [18]. This repository contains the source code for the Narrowbeer binary, the website to perform the attack on, a few licenses to reuse with Narrowbeer and the vulnerable version of the Widevine CDM. We also provide a comprehensive `README` file with instructions to perform the attack, on both Linux and Windows.

12 Ethics Considerations

We have timely reported our findings to the concerned parties, namely Google, Prime Video and Netflix. The three parties have acknowledged the effectiveness of the attack against their deployed systems. Since November 2024, Google Widevine has published a patch for our identified flaw, and revoked the vulnerable version. Thus, copyrighted content can no longer be pirated using our attack. Moreover, we only targeted streaming services with a clear responsible disclosure process. In order to prevent any leak, all experiments were performed on local resources under our full control.

References

[1] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *J. Netw. Comput. Appl.*, 48:44–57, 2015.

[2] Cataldo Basile, Bjorn De Sutter, Daniele Canavese, Leonardo Regano, and Bart Coppens. Design, implementation, and automation of a risk management approach for man-at-the-end software protection. *Comput. Secur.*, 132:103321, 2023.

[3] David Buchanan. MPEG-CENC: Defective by specification. *Phrack*, 16(71), August 2024.

[4] castr. OTT statistics & market insights. https://castr.com/blog/ott-statistics/, 2025.

[5] Sam Collins, Alex Poulopoulos, Marius Muench, and Tom Chothia. Anti-cheat: Attacks and the effectiveness of client-side defences. In *Checkmate@CCS*, pages 30–43. ACM, 2024.

[6] Stéphanie Delaune, Joseph Lallemand, Gwendal Patat, Florian Roudot, and Mohamed Sabt. Formal security analysis of widevine through the W3C EME standard. In *USENIX Security Symposium*. USENIX Association, 2024.

[7] Google. Shaka Player. https://github.com/shaka-project/shaka-player, 2025.

[8] Google. The EME Call and Event Logger Extension. https://github.com/shaka-project/eme_logger, 2025.

[9] Adam Gowdiak. Microsoft playready - complete client identity compromise. https://seclists.org/fulldisclosure/2024/May/5, 2021.

[10] Tomer Hadad. Reversing the old Widevine Content Decryption Module. https://github.com/tomer8007/widevine-l3-decryptor/wiki/Reversing-the-old-Widevine-Content-Decryption-Module, 2021.

[11] Gregory L. Heileman, Pramod A. Jamkhedkar, Joud S. Khoury, and Curtis J. Hrncir. The drm game. In *Proc. tth ACM Workshop on Digital Rights Management, (DRM'07)*, pages 54–62. ACM, 2007.

[12] Intel. Intel Pin. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, 2025.

[13] Tsuda Kageyu. MinHook. https://github.com/TsudaKageyu/minhook, 2024.

[14] NVIDIA. Fire It Up: Mozilla Firefox Adds Support for AI-Powered NVIDIA RTX Video. https://blogs.nvidia.com/blog/ai-decoded-rtxvideo-firefox/, 2024.

[15] Gwendal Patat, Mohamed Sabt, and Pierre-Alain Fouque. Exploring widevine for fun and profit. In *SP (Workshops)*, pages 277–288. IEEE, 2022.

[16] Gwendal Patat, Mohamed Sabt, and Pierre-Alain Fouque. Wideleak: How over-the-top platforms fail in android. In *DSN*, pages 501–508. IEEE, 2022.

[17] Gwendal Patat, Mohamed Sabt, and Pierre-Alain Fouque. Your DRM can watch you too: Exploring the privacy implications of browsers (mis)implementations of widevine EME. *Proc. Priv. Enhancing Technol.*, 2023(4):306–321, 2023.

[18] Florian Roudot and Mohamed Sabt. Artifact of the Narrowbeer Attack. https://doi.org/10.5281/zenodo.15525617.

[19] Statista. OTT video - united states. https://www.statista.com/outlook/amo/media/tv-video/ott-video/united-states#users, 2025.

[20] SysK. Practical cracking of white-box implementations. *Phrack*, 14(68), August 2012.

[21] Stephan van Schaik, Alexander Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Daniel Genkin, Andrew Miller, Eyal Ronen, Yuval Yarom, and Christina Garman. Sok: Sgx.fail: How stuff gets exposed. In *SP*, pages 4143–4162. IEEE, 2024.

[22] W3C. W3C Publishes Encrypted Media Extensions (EME) as a W3C Recommendation. https://www.w3.org/2017/09/pressrelease-eme-recommendation.html.en, 2017.

[23] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Steal this movie: Automatically bypassing DRM protection in streaming media services. In *USENIX Security Symposium*. USENIX Association, 2013.

[24] Qi Zhao. Wideshears: Investigating and breaking Widevine on QTEE. BlackHat Asia, 2021.

A The EME Workflow

An overview of the EME protocol is depicted in Figure 4. The starting point is the CDN, which provides more details about the encrypted content, including supported codecs and DRM systems. These details are used to access the desired CDM through the `requestMediaKeySystemAccess` method (not represented in Figure 4). The CDM is then instantiated using `createMediaKeys`, returning a `MediaKeys` object to access the CDM. Optionally, the initialization of the `MediaKeys` object can be completed by providing the certificate of the license server with the `setServerCertificate` method. Finally, a session is created within the CDM. The resulting `MediaKeySession` object is represented by a unique session ID and will be used to exchange data between the CDM and the license server. At this point, the CDM is initialized with a session. However, the key wallet of this session is empty and cannot decrypt any content.

The protocol goes forward; the decryption keys and corresponding usage policy, also known as licenses, must be loaded into the CDM. Thus, media-specific `Initialization Data` is sent to the CDM to generate a fresh license request. The content of this request is entirely dependent on the DRM system being used and is regarded as an opaque message within the EME framework. After receiving the license request, the EME user-agent forwards it to the license server. Note that although the user's authentication falls outside the scope of EME, most license servers will not respond to an unauthenticated user. The license server answers with an opaque license response containing the necessary keys to decrypt the protected content. The license response is sent to the CDM through the `update` method to be processed. A successful call to `update` means that the decryption keys have been loaded in the CDM memory; thereby, the HTML5 video element is ready to play the encrypted media.

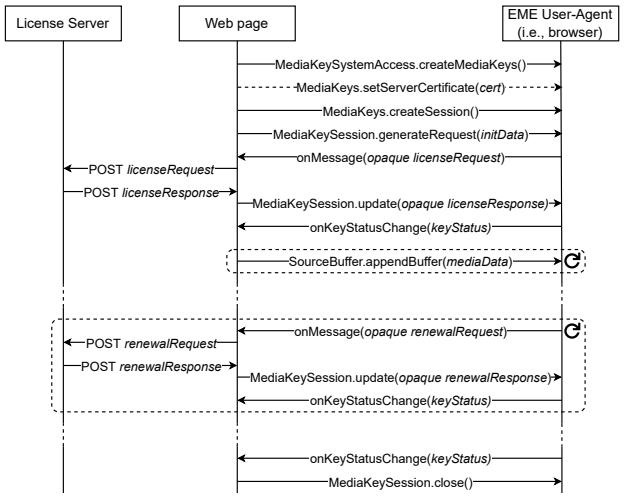

Figure 4: EME JavaScript Workflow

If the associated policy allows it, a license can be renewed. The renewal process only updates the license policies. If permitted, the CDM automatically generates a renewal request after a time specified by the license policy. Through the browser, this renewal request is sent to the license server, which answers with a renewal response that is transferred to the CDM through the same `update` method used previously.