Practically engageable adversaries for streaming media

Brian Sniffen

bsniffen@akamai.com
Akamai Technologies

Abstract. Streaming media content providers face a variety of adversaries. At the large scale at which they operate, they can justify defense in breadth: different techniques against different adversaries. Existing analysis techniques awkwardly handle questions about a distribution of adversaries against an evolving series of protocols. Contemporary streaming-media protocols eschew full-stream DRM in favor of token-based authentication at the start of each connection. Very recent protocols from several vendors package content in chunks of a few seconds each, rather than a stream continuing for an entire film or live event. Practical attackers work by copy and paste of these authentication tokens, sharing URLs to pay-per-view content in chat rooms or web forums. Others use "deep links" to chunks or lists of chunks to view freely available media in ways that violate the wishes of the providers. Accidental behaviors of so-called transparent proxies similarly violate the goals of the providers. We formalize these attacker descriptions in terms of an adversary who can only comprehend some tags in a tagged-concatenation Strand Spaces model. We use this model to show how current streaming protections work and fail, and suggest directions for new streaming media protocols.

1 Introduction

Millions of people use the Internet to engage with streaming media. One large provider, YouTube, reports billions of videos viewed per month [1]. Others (Hulu, NetFlix, Apple, Microsoft, etc.) operate at similar scale. These streaming media content providers have an unusual set of security goals. They are often happy to distribute content to anyone—or to a broad base of subscribers—but care that the content is not copied or presented outside its original context. Facing several distinct sorts of adversary, the content providers may make justified use of *defense in breadth.* They can specialize different defenses against different adversaries. Against some adversaries they may want technical protection using cryptographic protocols. Against other adversaries they may prefer social or legal protections. We propose *tag-limits* as a way to model the differences between these adversaries.

New developments in mobile computing and dynamic congestion response have led the dominant streaming media protocol developers to new designs. These "manifest-based" protocols complicate the problem of authentication and trust

To appear in FAST2009, pp. 1–9, 2009.
© Springer-Verlag Berlin Heidelberg 2009

in a distributed environment. They also provide a fine example to show how changes in protocol design can have different effects on adversaries with different tag limits.

Typical attacks against these streaming protocols do not exploit cryptographic flaws. They instead exploit information-flow weaknesses in the protocols: copy *this* authentication token into *that* URL and you can watch movies for free! This sort of problem is well-modeled by the Dolev-Yao adversary [2]. We model our protocols in the style of Thayer, Herzog, and Guttman's Strand Spaces [3] with rely-guarantee annotations for trust management [4]. This adversary may freely manipulate cryptographic encapsulations and may compose and decompose perfect concatenations of terms. The adversary is limited in that his concatenation operator is perfect—no bitstring can ever be subject to arbitrary parsing [5]. We further restrict our adversary: we distinguish concatenations by *tags*, then permit him only to compose or decompose certain tags. Tags should not be assumed to affect the wire format. Since any bitstring can be perfectly recognized as having a single parse, we feel free to assume perfect extraction of tags.

We will review the use of cryptographic protocols for streaming in Section 2, explaining the new manifest-based protocols in Section 2.3. We will show a sampling of security goals attributable to the major streaming protocol vendors in Section 3, then present models of the protocols in Section 4 and reasonable adversaries in Section 5. We will conclude by sketching advice for new streaming protocols in Section 6. Our principal contributions are the models of the adversaries and security goals in Sections 5 and 3, and descriptions of modern streaming protocols in Section 4.

2 Streaming basics

Traditional network streaming has worked with bespoke protocols over unreliable transit. A stream from a camera is translated to an efficient compressed format (e.g., MPEG4, Ogg Theora, H.264) at an *encoder*. A *server* is then given access to the encoded stream. For "video-on-demand" (VOD) streams, such as movies filmed in the past, the server will have a file on disk. For "live" streams showing real-time or near-real-time data, the server may itself receive the stream over the network. Clients connect to the server on some well-known control port and present a request for a stream to be sent to their own IP address. The server begins sending a sequence of unreliable datagrams to that address. The client receives most of them, decodes them, and presents a stream to the user.

Early clients were written as stand-alone programs (e.g., Real Media Player, Windows Media Player, QuickTime Player). Those quickly converted to browser plug-ins, each responsible for drawing a UI into a frame made available by the browser. Asynchronous updates to browsers, plug-ins, servers, and web pages made reliability a problem. Content providers had to rely on streaming protocol vendors not only for the protocol but for the art and polish of the user interface. Modern clients consist of three pieces:

Interpreter A browser plug-in that interprets some player. The Adobe Flash plugin, Silverlight Dynamic Language Runtime, and Objective-C runtime are interpreters.

Decoder Fast native code or accelerated hardware typically performs the actual H.264, MPEG, or Ogg Theora decoding.

Player An interpreted blob provides the UI and coördinates the connection between network, interpreter, decoder, and user. Adobe's SWF files are players in this sense, as are Microsoft's .NET-compiled objects.

Proprietary vendors typically have their own shuffling of these words. Some call the browser plug-in a player. Some call the decoder a player.

2.1 Classical streaming protocols

Even in the era of interpreted players, streaming media providers have mostly used bespoke protocols. The evolving bespoke protocols have continued to draw on public standard protocols. For example, several vendors have drawn on the successful Transaction Layer Security protocol (TLS) for inspiration for their own secret or authenticated session protocols. Few have preserved all the properties of TLS in so doing—but analysis of these protocols may be most revealing in a cryptographic context. We will not further discuss these protocols here.

More recent systems have used HTTP, the same protocol used for ordinary web data [6], to transfer the encoded stream.

2.2 URL and Cookie tokens

The principal authentication test used for access to these streams is a *token*. Tokens are typically keyed MACs computed over some easily-visible data. For our purposes, we can model these as encryptions to a key shared by all servers but no clients. For example, tokens might consist of encryptions over the date, the name of the stream requested, the IP of the client, or the entire rest of the request:

$$\{\!|\ 123.45.67.89, 26\ \text{Jul}\ 2009,\\ \texttt{http://www.example.com/...}|\!\}_K$$

For many reasons, IP addresses are often not included in the token: mobile devices often change address mid-stream and expect to be able to resume download at the same URL, many clients may share an address behind a NAT or proxy, and some clients may make each request in a sequence from a different address (as with multi-homed networks). In practice, therefore, a token typically does not include the client IP address:

$$\{\!|\ \text{26 Jul 2009 20:00,}\ \texttt{http://www.example.com/stream.mp4?}\ \texttt{date=200907262000}\,\&\,\texttt{timeout=5}\,|\!\}_K$$

One server can hand out such tokens, and another check that the token is properly assembled—in this example, that the enclosed date (26 Jul 2009 20:00) matches the date clearly in the URL (200907262000), that the current time is within 5 minutes (the "timeout" parameter) of the date shown, and that the stream URL requested exactly matches that shown in the token.

These tokens may be provided as part of the URL (e.g., appending `token=...` to the query string of the URL) or as a cookie set in the browser and sent as a separate header. As we will see later, these choices enable or combat different adversaries.

2.3 Manifests

HTTP is well suited for downloads of small (or at least finite) objects. So-called "progressive-download" extensions to HTTP can help with large and unending files—but typical solutions end up using query parameters to express offsets into a file, then leaving an HTTP connection open for a very long time. Because they use TCP, it is awkward to switch bit-rates or offsets within a stream—as when changing networks or when a user skips to a different part of a movie. One solution is to close the stream and request a new stream, accepting the cost to smooth user experience.

Another solution puts control in the client's court and fits better with design of HTTP: break a stream into many small components, providing the client with a *manifest* of these components [7]. Live streams may then use progressive download, refresh/pull, or other mechanisms to keep the client's view of the manifest fresh.

Authentication tokens may be embedded into the request for a manifest or into the URLs inside the manifest. Cookie tokens may be set, cleared, or updated with each request. The first few lines of a manifest without any tokens embedded is shown in Fig. 1. Each line represents a link to a few seconds of content.

3 Content provider security goals

The principal actors choosing and deploying streaming media protocols are *content providers.* These are typically not copyright holders for the media they serve; nor do they typically operate servers which directly service users. Rather, they operate sites like Hulu, YouTube, or NetFlix. These content providers serve as a nexus to license rights from rights-holders, purchase storage and content delivery services, and provide a central brand identity for users.

These content providers' goals are distinct from those of the rights-holders, though they must also satisfy the rights-holders security policies to remain in business.

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-MEDIA-SEQUENCE:176707
#EXTINF:10,
http://example.com/example/fileSequence176707.ts
#EXTINF:10,
http://example.com/example/fileSequence176708.ts
#EXTINF:10,
http://example.com/example/fileSequence176709.ts
#EXTINF:10,
http://example.com/example/fileSequence176710.ts
```

Fig. 1. Example manifest

3.1 Copying and DRM

For purposes of this analysis, we will assume that any general purpose computer displaying a video or playing an audio stream may save a copy, and may later distribute that copy. We will not further address issues of copying or restrictions on copying, including rights-management software, save to note that high-quality media files are quite large, and have grown with the growing availability of big disks and wide pipes. We can treat those adversaries with the resources to copy and re-distribute media very differently from the adversaries below.

3.2 Deep linking

The principal security goal we consider is the control of access to and presentation of the content. Our content providers are specifically concerned with preventing various forms of "deep linking[1]."

Content providers often hand out URLs to streams for free, but only after a short ceremony (e.g., watching an advertisement or registering an e-mail address). They do not want these URLs shared over chat-rooms (which provide only recently-posted URLs) or web forums (which may archive many URLs). They are already comfortable using rapidly expiring authentication tokens to achieve this goal in pre-manifest systems. The technique ports cleanly to protect the manifest itself. We will discuss how to extend this technique to the contents of a manifest in Section 6.

Other content providers may also act adversarially. For example, one content provider might be happy to show its own ads and receive revenue for them, then direct users to pull a stream from a victim content provider. It also might do this by linking directly to a pre-manifest stream, by linking to a manifest, or by providing its own manifest linking to the stream components.

[1] Not to be confused with the laudable practice of removing `<blink>` tags from the Web.

All of these security goals are instances of the problem of distributed authentication and session management developed by Saltzer and Schroeder, and separately by Abadi, Burrows, and Lampson.

3.3 Bad player

The interpreted player provides advertising, theming, branding, skinning, quality control, and other forms of user interface management. The content providers want to ensure that the media stream is presented in the most recent player—not merely any authentic player, but the currently released player.

4 Protocols

We represent our protocols with usual Strand Space notation. We show concatenation of A and B as $A \circ B$. We show tagged concatenation as $A \circ_{\text{tag}} B$. All servers share a secret key K. The client and a web server share a TLS session key K_S.

We first examine a standard old-style streaming media protocol, shown in Fig. 2. A client receives a URL from a web server over a confidential connection (perhaps TLS). The client sends that URL to a media server, which responds with many packets. Each packet is a small amount of stream content. The URL passed around has several security-relevant objects embedded within it: release of the URL by the client shows a grant of permission from the web server to receive a stream identified by base-URL within 5 minutes of 26-Jul-09. The last element of the URL is an embedded authentication token. It shows that some server knowing K granted some client this permission. Only adversaries capable of manipulating the tag URL can project values from this message or construct messages of this form.

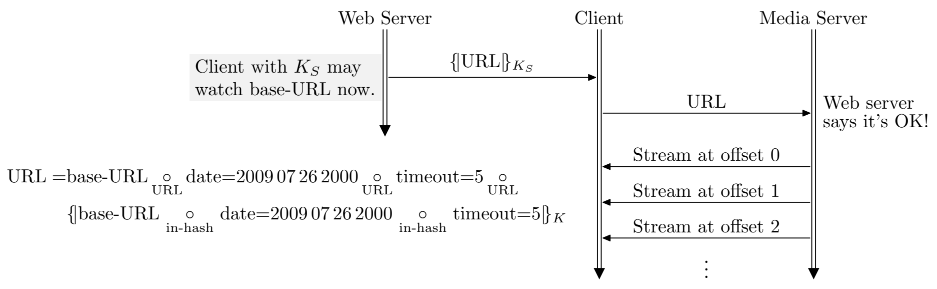
Web Server
Client
Media Server
Client with K_S may watch base-URL now.
$\{\!|\text{URL}|\!\}_{K_S}$
URL
Web server says it's OK!
Stream at offset 0
Stream at offset 1
Stream at offset 2
⋮
URL =base-URL $\circ_{\text{URL}}$ date=2009 07 26 2000 $\circ_{\text{URL}}$ timeout=5 $\circ_{\text{URL}}$
$\{\!|$base-URL $\circ_{\text{in-hash}}$ date=2009 07 26 2000 $\circ_{\text{in-hash}}$ timeout=5$|\!\}_K$

Fig. 2. Simple streaming protocol

We now turn to a manifest-based protocol in Fig. 3. The web server makes the same initial trust decision before releasing a URL to the client. The client then requests that URL's resource from the media server. The media server responds with a manifest: a list of URLs. The client then requests those in some order. It may pre-fetch some before watching. It may skip some in response to user commands or network congestion.

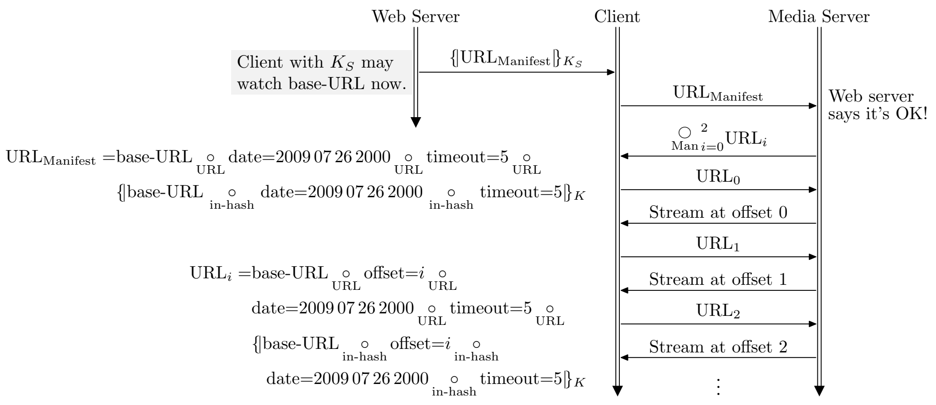
Web Server
Client
Media Server
Client with K_S may watch base-URL now.
$\{|\text{URL}_{\text{Manifest}}|\}_{K_S}$
$\text{URL}_{\text{Manifest}}$
Web server says it's OK!
$\bigcirc^{2}_{\text{Man } i=0}\text{URL}_i$
$\text{URL}_{\text{Manifest}} = \text{base-URL} \underset{\text{URL}}{\circ} \text{date=2009 07 26 2000} \underset{\text{URL}}{\circ} \text{timeout=5} \underset{\text{URL}}{\circ}$
$\{|\text{base-URL} \underset{\text{in-hash}}{\circ} \text{date=2009 07 26 2000} \underset{\text{in-hash}}{\circ} \text{timeout=5}|\}_K$
URL_0
Stream at offset 0
URL_1
$\text{URL}_i = \text{base-URL} \underset{\text{URL}}{\circ} \text{offset=}i \underset{\text{URL}}{\circ}$
$\text{date=2009 07 26 2000} \underset{\text{URL}}{\circ} \text{timeout=5} \underset{\text{URL}}{\circ}$
$\{|\text{base-URL} \underset{\text{in-hash}}{\circ} \text{offset=}i \underset{\text{in-hash}}{\circ}$
$\text{date=2009 07 26 2000} \underset{\text{in-hash}}{\circ} \text{timeout=5}|\}_K$
Stream at offset 1
URL_2
Stream at offset 2
⋮

Fig. 3. Manifest-based streaming protocol

What sort of token should we embed in the chunk URLs? The diagram shows a fresh token for each chunk, which will run into problems with timeouts during long streams:

$$\{|\text{base-URL} \underset{\text{in-hash}}{\circ} \text{offset=}i \underset{\text{in-hash}}{\circ} \text{date=26-Jul-09} \underset{\text{in-hash}}{\circ} \text{timeout=5}|\}_K$$

We might use the same token as the manifest URL:

$$\{|\text{base-URL} \underset{\text{in-hash}}{\circ} \text{date=26-Jul-09} \underset{\text{in-hash}}{\circ} \text{timeout=5}|\}_K$$

Again, this will have problems with long streams. By using cookie-based tokens instead, we might consider chaining tokens from one request to the next, weaving access control decisions through the protocol. We might even consider using no token at all. Depending on how we model the unreliable datagrams of pre-manifest protocols and the reliable channels of TCP, we might draw several different conclusions about the consequences of these different tokens in the presence of various adversaries.

Because manifests of live streams have messages of unbounded length, we do not attempt to address them in the strand spaces formalism. We hope that the uniform message structure will allow a manual proof.

5 Adversaries

Classical strand spaces define the following adversary actions, each shown as a short sequence of actions, where $-a$ indicates receiving a value called a and $+b$ indicates sending that value:

Fresh texts $\mathsf{M}_t : \langle +t \rangle$ where $t \in$ text
Concatenation $\mathsf{C}_{g,h} : \langle -g, -h, +g \circ h \rangle$
Encryption $\mathsf{E}_{h,K} : \langle -K, -h, +\{\!|h|\!\}_K \rangle$
Fresh keys $\mathsf{K}_K : \langle +K \rangle$
Selection $\mathsf{S}_{g,h} : \langle -g \circ h, +g, +h \rangle$
Decryption $\mathsf{D}_{h,K} : \langle -K^{-1}, -\{\!|h|\!\}_K, +h \rangle$.

Our adversary does have $\mathsf{C}_{g,h}$ and $\mathsf{S}_{g,h}$ for assembling and disassembling untagged concatenations, but also has the ability to manipulate some tagged concatenations T_A:

$$\mathsf{C}_{g,h,t} : \langle -g, -h, +g \underset{t}{\circ} h \text{ where } t \in T_A \rangle$$
$$\mathsf{S}_{g,h,t} : \langle -g \underset{t}{\circ} h, +g, +h \text{ where } t \in T_A \rangle$$

We can now define a tagged strand space Σ_{T_A} as a strand space Σ annotated with a set of tags T_A that can be manipulated by the adversary.

We can now write down the adversaries to the security goals mention in Section 3. The simplest copy-and-paste adversary cannot manipulate any tagged concatenations: $T_A = \{\}$. He can only replay entire messages. This also well models the accidental adversary, such as a mis-configured HTTP proxy. Adversaries who can manipulate URLs but do not understand the manifest format have $T_A = \{\text{URL}\}$. Adversaries who can make manifest files but cannot manipulate URLs seem unrealistic, but would have $T_A = \{\text{Man}\}$. Adversaries similar to the competing content providers of Section 3.2 are quite flexible. They have $T_A = \{\text{Man}, \text{URL}\}$. Of course, with bank accounts and some need to build a brand, these flexible adversaries may be more vulnerable to social controls.

6 Future Work

We can now analyze these protocols, including the several token variants mentioned in Section 4, in light of the various adversaries from Section 5. We expect to find a set of token features necessary to protect against $T_A = \{Man, URL\}$ adversaries given reasonable assumptions about the TCP channel between client and media server and stateful servers. With stateless servers, we have little hope to protect against $T_A \supset \{Man\}$ adversaries—even regular clients will connect to several different servers during a stream. As they rewind and fast-forward, their behavior will be indistinguishable from adversary behavior to the server.

We can also explore the problem of Interpreter-verified Players. Recall these definitions from Section 2. Several streaming protocol vendors hope to (in our terms) prevent a tag-limited adversary from using secrets embedded in a player other than in accordance with the protocol. These adversaries cannot manipulate the tag binding the secret to the Player.

We also see work to justify the model of the adversary as tag-limited, and to refine an appropriate model of TCP, UDP, and IP for Strand Spaces.

7 Acknowledgments

My thanks to Michael Stone, Katherine Sniffen, and Andy Ellis for helpful comments on drafts of this work.

References

1. comScore: YouTube surpasses 100 million U.S. viewers for the first time (3 2009)
2. Dolev, D., Yao, A.C.: On the security of public-key protocols. In: IEEE Transactions on Information Theory. Volume 2. (1983) 198–208
3. Thayer, F., Herzog, J.C., Guttman, J.D.: Strand spaces: Proving security protocols correct. Journal of Computer Security **7**(1) (1999)
4. Guttman, J.D., Thayer, F.J., Carlson, J.A., Herzog, J.C., Ramsdell, J.D., Sniffen, B.T.: Trust management in strand spaces: A rely-guarantee method. In: In Proc. of the European Symposium on Programming (ESOP 2004), LNCS, Springer-Verlag (2004) 325–339
5. Boyd, C.: Hidden assumptions in cryptographic protocols. IEE Proceedings Computers and Digital Techniques **137**(6) (November 1990) 433–436
6. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard) (June 1999) Updated by RFC 2817.
7. Zambelli, A.: IIS smooth streaming overview (3 2009)