WEBPUSH
Internet-Draft
Intended status: Standards Track
Expires: September 7, 2015

E. Damaggio
B. Raymor
Microsoft

March 6, 2015

Generic Event Delivery Using HTTP Push
draft-damaggio-webpush-http2-00

Abstract

A simple protocol for the delivery of realtime events to user agents is described. This scheme uses HTTP/2 server push.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 7, 2015.

Table of Contents

1. Introduction

Many applications on mobile and embedded devices require continuous access to network communications so that real-time events - such as incoming calls or messages - can be conveyed (or "pushed") in a timely fashion.

Mobile and embedded devices typically have limited power reserves, so finding more efficient ways to serve application requirements greatly benefits the application ecosystem. One significant contributor to power usage is the radio. Radio communications consume a significant portion of the energy budget on a wirelessly connected device.

Uncoordinated use of persistent connections or sessions can contribute to unnecessary use of the device radio, since each independent session independently incurs overheads. In particular, keep alive traffic used to ensure that middleboxes do not prematurely time out sessions, can result in significant waste. Maintenance

traffic tends to dominate over the long term, since events are relatively rare.

Consolidating all real-time events into a single session ensures more efficient use of network and radio resources. A single service consolidates all events, distributing those events to applications as they arrive. This requires just one session, avoiding duplicated overhead costs.

A push server that does not support reliable delivery over intermittent network connections or failing applications on devices, forces the device to acknowledge receipt directly to the application server, incurring additional power drain in order to establish (usually secure) connections to the individual application servers.

While reliability is not required for messages that expire in few seconds (e.g. an incoming call) or collapsible ones (e.g. the current number of unread emails), it is more important when messages contain information that is longer lasting, e.g. a command to update a configuration value, or the acknowledgement of a financial transaction or workflow step. In particular, in the case of power-constrained devices, it is preferable to transmit the actual information in the "pushed" message reliably, instead of forcing them to reconnect periodically to get the current state.

An open standard to "push" messages to embedded and mobile devices:

- Simplifies deployment of "push" features across a variety of mobile and embedded device platforms
- Creates an ecosystem of services (e.g. consolidation services) and development tools enabling efficient "push"
- Technically enables consolidating real-time events into a single session which is impossible when each "push" implementation is built in isolation

There are two primary scenarios under consideration:

- Web applications in a mobile user agent and
- Embedded devices receiving push messages from cloud services through an intermediate "field gateway", i.e. a reasonably powerful device (capable of secure HTTP/2 communications), which acts as a local agent.

The W3C Web Push API [API] describes an API that enables the use of a consolidated push service from web applications. This expands on that work by describing a protocol that can be used to:

- request the delivery of a push message to a user agent,
- create new push message delivery subscriptions, and
- monitor for new push messages.

Requesting the delivery of events is particularly important for the Web Push API. The registration, management and monitoring functions are currently fulfilled by proprietary protocols; these are adequate, but do not offer any of the advantages that standardization affords.

In the embedded field gateway scenario, small (possibly much less capable devices) connect to a field gateway to receive push messages. This protocol does not detail the device-to-field gateway connection, instead it details how the field-gateway can efficiently receive push messages on behalf of many devices.

This document intentionally does not describe how a push service is discovered. Discovery of push services is left for future efforts, if it turns out to be necessary at all. User agents are expected to be configured with a URL for one (or more) push services.

1.1. Conventions and Terminology

In cases where normative language needs to be emphasized, this document falls back on established shorthands for expressing interoperability requirements on implementations: the capitalized words "MUST", "MUST NOT", "SHOULD" and "MAY". The meaning of these is described in [RFC2119].

This document defines the following terms:

application: Both the sender and ultimate consumer of push messages. Many applications have components that are run on a user agent and other components that run on servers.

application server: The component of an application that runs on a server and requests the delivery of a push message.

push message: A message, sent from an application server to a user agent via a push service.

push service: A service that delivers push messages to user agents.

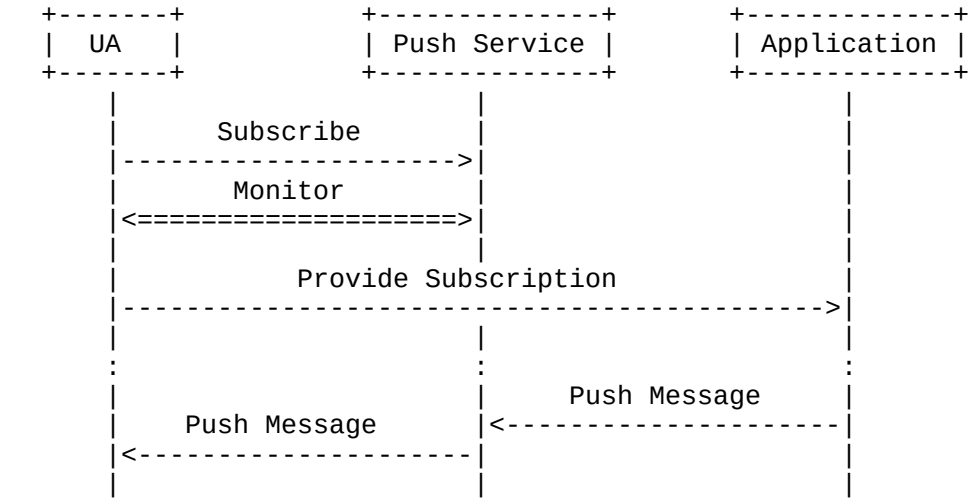
subscription: A message delivery context that is established between the user agent and the push service and shared with applications. All push messages are associated with a subscription.

user agent: A device and software that is the recipient of push messages.

Examples in this document use the HTTP/1.1 message format [RFC7230]. Many of the exchanges can be completed using HTTP/1.1, where HTTP/2 is necessary, the more verbose frame format from [I-D.ietf-httpbis-http2] is used.

2. Overview

A general model for push services includes three basic actors: a user agent, a push service, and an application (server).



At the very beginning of the process, a new subscription is created by the user agent and then distributed to an application server. The subscription is the basis of all future interactions between the user agent and push service.

It is expected that a different subscription will be provided to each application; however, there are no inherent cardinality constraints in the protocol. Multiple subscriptions might be created for the same application, or multiple applications could use the same subscription. Note however that sharing subscriptions can have security and privacy implications.

Application servers use subscriptions to deliver push messages to user agents, via the push service.

Subscriptions have a limited lifetime. They can also be terminated by either push service or user agent at any time. User agents and application servers need to be prepared to manage changes in subscription state.

2.1. HTTP Resources

This protocol uses HTTP resources [RFC7230] and link relations [RFC5988]. The following resources are defined:

push service: This resource is used in Subscribing (Section 4). It is configured into user agents.

subscription: A link relation of type "urn:ietf:params:push" refers to a subscription resource. Subscription resources are used to deliver push messages. An application server sends push messages (Section 5) and a user agent receives push messages (Section 6) using this resource.

receipt: A link relation of type "urn:ietf:params:push:receipt" refers to a delivery receipt resource. An application server receives delivery confirmation (Section 5.1) for push messages using this resource.

3. Registration

The Registration and Subscribe resources referenced in [I-D.draft-thomson-webpush-http2-02] were deprecated to eliminate the overhead of maintaining registration-subscription relationships in the push server.

4. Subscribing

A user agent sends a POST request to its configured push service resource to create a new subscription.

```
POST /subscribe/ HTTP/1.1
Host: push.example.net
```

A response with a 201 (Created) status code includes a URI for the subscription in the Location header field.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Link: </p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx>;
        rel="urn:ietf:params:push"
Location: https://push.example.net/p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx
Cache-Control: max-age:864000, private
```

The user agent should securely distribute the "subscription" resource to its application server. (Details are outside the scope of this document.)

5. Requesting Push Message Delivery

An application server requests the delivery of a push message by sending an HTTP POST request to the "subscription" resource distributed by its user agent. The push message is included in the body of the request.

The push message is a JSON [RFC7159] object which contains the push message data and directives for the push server:

Member	Use	Description
message	optional	A JSON object that contains push message data
request_receipt	optional	A JSON boolean indicating whether the application server requests a confirmation that the push message was delivered to the user agent. Its default value is false.
time_to_live	optional	A JSON number that represents the expiration time in seconds for a push message. It is relative to the time that the push server receives the request. A message MUST NOT be delivered after it expires.

Table 1: Push Message Request Format

```
POST /p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx HTTP/1.1
Host: push.example.net
Content-Type: application/json
Content-Length: ...

{
 "request_receipt": true,
 "message": {"data": "Hello World"}
}
```

A response with a 201 (Created) status code includes a URI for the message in the Location header field. This does not indicate that the message was delivered to the user agent. If a receipt was requested, then a URI for the receipt resource is included in the Link header field in the response.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:55 GMT
Link: </r/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx/;
        rel="urn:ietf:params:push:receipt"
Cache-Control: max-age=600
Location: https://push.example.net/p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx/id
```

A push server MUST return a 400 (Bad Request) status code in response to a POST request that contains malformed JSON in the body.

[Should the push server return a 400 if the requested time_to_live exceeds its storage limits?]

A push service MAY generate a 413 (Payload Too Large) status code in response to POST requests that include an entity body that is too large. Push services MUST NOT generate a 413 status code in responses to an entity body that is 4k (4096 bytes) or less in size.

5.1. Requesting Push Message Receipts

The application server MAY request to be notified by the push server when a push message has been successfully delivered to the user agent.

To request a receipt, the application server sets the value of the push message request_receipt member to true in the HTTP POST request to the "subscription" resource.

The application server requests the delivery of receipts from the push server by making a HTTP GET request to the "receipt" resource. The push service does not respond to this request, it instead uses

HTTP/2 server push [I-D.ietf-httpbis-http2] to send the content of push receipts when messages are acknowledged by the user agent.

[Details on the message format for push receipt responses is TBD]

The push server MUST generate a 504 (Gateway Timeout) if the user agent fails to acknowledge the receipt of the push message or the push server fails to deliver the message prior to its expiration.

6. Receiving Push Messages

A user agent requests the delivery of new push messages by making a GET request to the "subscription" resource. The push service does not respond to this request, it instead uses HTTP/2 server push [I-D.ietf-httpbis-http2] to send the contents of push messages as they are sent by application servers.

Each push message is pushed in response to a synthesized GET request. The GET request is made to the "subscription" resource. The response body is the entity body from the most recent POST request sent to the "subscription" resource by the application server.

The following example request is made over HTTP/2.

```
HEADERS      [stream 7] +END_STREAM +END_HEADERS
  :method        = GET
  :path          = /p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx
  :authority     = push.example.net
```

The push service permits the request to remain outstanding. When a push message is sent by an application server, a server push is associated with the initial request. The response includes the push message.

```
PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
  :method        = GET
  :path          = /p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx/id
  :authority     = push.example.net

HEADERS      [stream 4] +END_HEADERS
  :status        = 200
  date           = Thu, 11 Dec 2014 23:56:55 GMT
  last-modified  = Thu, 11 Dec 2014 23:56:55 GMT
  cache-control  = private
  content-type   = ...
  content-length = ...

DATA         [stream 4] +END_STREAM
  { // JSON Object // }
```

A user agent might receive a PUSH_PROMISE for a resource for which it has no active subscription. The resulting unwanted push message can be ignored, or the corresponding stream can be reset (using RST_STREAM) to avoid expending bandwidth.

A user agent can request the contents of the "subscription" resource immediately by including a Prefer header field [RFC7240] with a "wait" parameter set to "0". The push server SHOULD return a link reference to the "subscription" resource and expiration information in response to this request. This request also triggers the delivery of all push messages that the push service has stored but not yet delivered. The server MUST generate a server push for all stored messages that have not yet been delivered.

Different collapsing or coalescing disciplines for messages are possible but outside the scope of this document.

6.1. Acknowledging Push Message Receipts

To enable "at least once delivery", the user agent MUST acknowledge receipt of the message by performing a HTTP DELETE on the resource in the :path pseudo-header field from the PUSH_PROMISE.

```
DELETE /p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx/id HTTP/1.1
Host: push.example.net
```

If the application has requested a delivery receipt, the push server MUST deliver a response to the application server monitoring the "receipt" resource.

7. Operational Considerations

[No changes to [I-D.draft-thomson-webpush-http2-02]]

7.1. Load Management

[No changes to [I-D.draft-thomson-webpush-http2-02]]

7.2. Push Message Expiration

[This section from [I-D.draft-thomson-webpush-http2-02] was updated to include the time_to_live option.]

Push services typically store messages for some time to allow for limited recovery from transient faults. If a push message is stored, but not delivered, the push service can indicate the probable duration of storage by including expiration information in the response to the push request.

A push service is not obligated to store messages indefinitely. If a user agent is not actively monitoring for push messages, those messages can be lost or overridden by newer messages on the same subscription.

Push messages that were stored and not delivered to a user agent are delivered when the user agent recommences monitoring. (A message with a time_to_live option MUST NOT be delivered once it expires.) Stored push messages SHOULD include a Last-Modified header field (see Section 2.2 of [RFC7232]) indicating when delivery was requested by an application server.

A GET request to a "subscription" resource that has expired messages results in a 204 (No Content) status response, equivalent to if no push message were ever sent.

Push services might need to limit the size and number of stored push messages to avoid overloading. In addition to using the 413 (Payload Too Large) status code for too large push messages, a push service MAY expire push messages prior to any advertised expiration time.

7.3. Subscription Expiration

[Minor editorial changes to [I-D.draft-thomson-webpush-http2-02] to remove references to registration]

In some cases, it may be necessary to terminate subscriptions so that they can be refreshed.

A push service might choose to set a fixed expiration time. If a resource has a known expiration time, expiration information is included in responses to requests that create the resource, or in requests that retrieve a representation of the resource.

Expiration is indicated using either the Expires header field, or by setting a "max-age" parameter on a Cache-Control header field (see [RFC7234]). The Cache-Control header field MUST also include the "private" directive.

A push service can invalidate a subscription at any time. If a user agent has an outstanding request to the "subscription" resource, this can be signaled by returning a 400-series status code, such as 410 (Gone). A push service uses server push to indicate that a subscription has expired; a pushed 400-series status code for the subscription resource signals the termination of a subscription.

A user agent can request that a subscription be removed by sending a DELETE request to the corresponding URI.

A push service MUST send a 400-series status code, such as 404 (Not Found) or 410 (Gone) if an application server atttempts to send a push message to a removed or expired subscription.

7.4. Implications for Application Reliability

[This section from [I-D.draft-thomson-webpush-http2-02] was updated to include receipts.]

The availability of push message delivery receipts in the protocol ensures that the application developer is not tempted to create alternative mechanisms for message delivery in case the push service fails to deliver a critical message. Setting up a polling mechanism or a backup messaging channel in order to compensate for these shortcomings negates almost all of the advantages a push service provides.

8. Security Considerations

[Minor editorial changes throughout Section 8 to [I-D.draft-thomson-webpush-http2-02] to remove references to registration]

This protocol MUST use HTTP over TLS [RFC2818]; this includes any communications between user agent and push service, plus communications between the application and the push service. Thus, all URIs use the "https" scheme. This provides confidentiality and integrity protection for subscriptions and push messages from external parties.

8.1. Confidentiality from Push Service Access

The protection afforded by TLS does not protect content from the push service. Without additional safeguards, a push service is able to see and modify the content of the messages.

Applications are able to provide additional confidentiality, integrity or authentication mechanisms within the push message itself. The application server sending the push message and the application on the user agent that receives it are frequently just different instances of the same application, so no standardized protocol is needed to establish a proper security context. The process of providing the application server with subscription information provides a convenient medium for key agreement.

The Web Push API codifies this practice by requiring that each push subscription created by the browser be bound to a browser generated encryption key. Pushed messages are authenticated and decrypted by the browser before delivery to applications. This scheme ensures that the push service is unable to examine the contents of push messages.

The public key for a subscription ensures that applications using that subscription can identify messages from unknown sources and discard them. This depends on the public key only being disclosed to entities that are authorized to send messages on the channel. The push server does not require access to this public key.

8.2. Privacy Considerations

Push message confidentiality does not ensure that the identity of who is communicating and when they are communicating is protected. However, the amount of information that is exposed can be limited.

Subscription URIs MUST NOT provide any basis to correlate communications for a given user agent. It MUST NOT be possible to correlate any two subscription URIs based solely on the content of the subscription URIs. This allows a user agent to control correlation across different applications, or over time.

In particular, user and device information MUST NOT be exposed through the subscription URI.

In addition, subscription URIs established by the same user agent MUST NOT include any information that allows them to be correlated with the associated user agent.

Note: This need not be perfect as long as the resulting anonymity set (see [RFC6973], Section 6.1.1) is sufficiently large. A subscription URI necessarily identifies a push service or a single server instance. It is also possible that traffic analysis could be used to correlate subscriptions.

A user agent MUST be able to create new subscriptions with new identifiers at any time.

8.3. Authorization

This protocol does not define how a push service establishes whether a user agent is permitted to create a subscription, or whether push messages can be delivered to the user agent. A push service MAY choose to authorize request based on any HTTP-compatible authorization method available, of which there are numerous options. The authorization process and any associated credentials are expected to be configured in the user agent along with the URI for the "push service".

Authorization for sending push messages is managed using capability URLs (see [CAP-URI]). A capability URL grants access to a resource based solely on knowledge of the URL. Capability URLs are used for their "easy onward sharing" and "easy client API" properties. These make it possible to avoid relying on relationships between push services and application servers, with the protocols necessary to build and support those relationships.

A subscription URI therefore acts as a bearer token: knowledge of the URI implies authorization to send push messages. Subscription URIs MUST be extremely difficult to guess. Encoding a large amount of random entropy (at least 120 bits) in the path component ensures that it is difficult to successfully guess a valid subscription URI.

8.4. Denial of Service Considerations

Discarding unwanted messages at the user agent based on message authentication doesn't protect against a denial of service attack on the user agent. Even a relatively small volume of push messages can cause battery-powered devices to exhaust power reserves. Limiting the number of entities with access to push channels limits the number of entities that can generate value push requests of the push server.

An application can limit where push messages can originate by limiting the distribution of subscription URIs to authorized entities. Ensuring that subscription URIs are hard to guess ensures that only applications servers that have been given a subscription URI can use it.

A malicious application with a valid subscription use the greater resources of a push service to mount a denial of service attack on a user agent. Push service SHOULD limit the rate at which push messages are sent to individual user agents. A push service or user agent MAY terminate subscriptions (Section 7.3) that receives too many push messages.

Conversely, a push service is also able to deny service to user agents. Intentional failure to deliver messages is difficult to distinguish from faults, which might occur due to transient network errors, interruptions in user agent availability, or genuine service outages.

8.5. Logging Risks

Server request logs can reveal subscription URIs. Acquiring a subscription URI permits the sending of push messages. Logging could also reveal relationships between different subscription URIs for the same user agent.

Limitations on log retention and strong access control mechanisms can ensure that URIs are not learned by unauthorized entities.

9. IANA Considerations

This document registers XXXXX URNs for use in identifying link relation types. These are added to a new "Web Push Identifiers" registry according to the procedures in Section 4 of [RFC3553]; the corresponding "push" sub-namespace is entered in the "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry.

The "Web Push Identifiers" registry operates under the IETF Review policy [RFC5226].

Registry name: Web Push Identifiers

URN Prefix: urn:ietf:params:push

Specification: (this document)

Respository: [Editor/IANA note: please include a link to the final registry location.]

Index value: Values in this registry are URNs or URN prefixes that start with the prefix "urn:ietf:params:push". Each is registered independently.

New registrations in the "Web Push Identifiers" are encouraged to include the following information:

URN: A complete URN or URN prefix.

Description: A summary description.

Specification: A reference to a specification describing the semantics of the URN or URN prefix.

Contact: Email for the person or group making the registration.

Index value: As described in [RFC3553], URN prefixes that are registered include a description of how the URN is constructed. This is not applicable for specific URNs.

Two values are entered as the initial content of the "Web Push Identifiers" registry.

URN: urn:ietf:params:push

Description: This link relation type is used to identify a web push subscription.

Specification: (this document)

Contact: Web Push WG (webpush@ietf.org)

URN: urn:ietf:params:push:receipt

Description: This link relation type is used to identify a resource for receiving delivery receipts for push messages.

Specification: (this document)

Contact: Web Push WG (webpush@ietf.org)

10. Acknowledgements

This document incorporates and iterates on material from [I-D.draft-thomson-webpush-http2-02].

11. References

11.1. Normative References

[CAP-URI] Tennison, J., "Good Practices for Capability URLs", FPWD capability-urls, February 2014, <http://www.w3.org/TR/capability-urls/>.

[I-D.draft-thomson-webpush-http2-02] Thomson, M., "Generic Event Delivery Using HTTP Push (work in progress)", December 2014, <https://tools.ietf.org/html/draft-thomson-webpush-http2-02.txt>.

[I-D.ietf-httpbis-http2] Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol version 2", draft-ietf-httpbis-http2-17 (work in progress), February 2015.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, June 2003.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.

[RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

[RFC7230] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

[RFC7232] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, June 2014.

[RFC7234] Fielding, R., Nottingham, M., and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, June 2014.

[RFC7240] Snell, J., "Prefer Header for HTTP", RFC 7240, June 2014.

11.2. Informative References

[API] Sullivan, B. and E. Fullea, "Web Push API", ED push-api, December 2014, <https://w3c.github.io/push-api/>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, July 2013.

Authors' Addresses

Elio Damaggio
Microsoft
One Microsoft Way
Redmond, WA 98052
US

Email: elioda@microsoft.com

Brian Raymor
Microsoft
One Microsoft Way
Redmond, WA 98052
US

Email: brian.raymor@microsoft.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 10, 2015

M. Thomson
Mozilla

March 9, 2015

Message Encryption for Web Push
draft-thomson-webpush-encryption-00

Abstract

A message encryption scheme is described for the Web Push protocol. This scheme provides confidentiality and integrity for messages sent from an Application Server to a User Agent.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

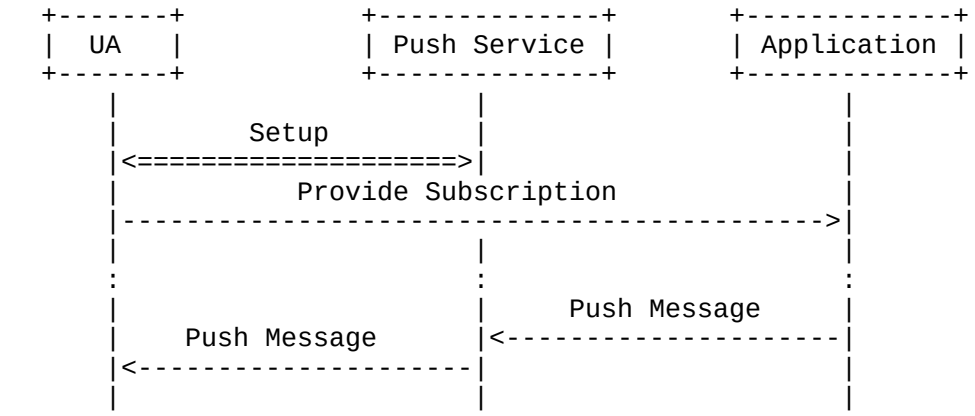
This Internet-Draft will expire on September 10, 2015.

Copyright Notice

Table of Contents

1. Introduction

The Web Push protocol [I-D.thomson-webpush-http2] is an intermediated protocol by necessity. Messages from an Application Server are delivered to a User Agent via a Push Service.



This document describes how messages sent using this protocol can be secured against inspection or modification by a Push Service.

Web Push messages are the payload of an HTTP message [RFC7230]. These messages are encrypted using an encrypted content encoding [I-D.nottingham-http-encryption-encoding]. This document describes how this content encoding is applied and describes a recommended key management scheme.

For efficiency reasons, multiple users of Web Push often share a central agent that aggregates push functionality. This agent can enforce the use of this encryption scheme by applications that use push messaging. An agent that only delivers messages that are properly encrypted strongly encourages the end-to-end protection of messages.

A web browser that implements the Web Push API [API] can enforce the use of encryption by forwarding only those messages that were properly encrypted.

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It’s not shouting, when they are capitalized, they have the special meaning described in [RFC2119].

2. Key Generation and Agreement

For each new subscription that the User Agent generates for an application, it also generates an asymmetric key pair for use in Diffie-Hellman (DH) [DH] or elliptic-curve Diffie-Hellman (ECDH) [ECDH]. The public key for this key pair can then be distributed by the application to the Application Server along with the URI of the subscription. The private key MUST remain secret.

This key pair is used with the Diffie-Hellman key exchange as described in Section 4.2 of [I-D.nottingham-http-encryption-encoding].

A User Agent MUST generate and provide a public key for the scheme described in Section 5.

The public key MUST be accompanied by a key identifier that can be used in the "keyid" parameter to identify which key is in use. Key identifiers need only be unique within the context of a subscription.

2.1. Diffie-Hellman Group Information

As described in [I-D.nottingham-http-encryption-encoding], use of Diffie-Hellman for key agreement requires that the receiver provide clear information about it’s chosen group and the format for the "dh" parameter with each potential sender.

This document only describes a single ECDH group and point format, described in Section 5. A specification that defines alternative groups or formats MUST provide a means of indicating precisely which group and format is in use for every public key that is provided.

2.2. Key Distribution

The application using the subscription distributes the key identifier and public key along with other subscription information, such as the subscription URI and expiration time.

The communication medium by which an application distributes the key identifier and public key MUST be confidentiality protected for the reasons described in [I-D.thomson-webpush-http2]. Most applications that use push messaging have a pre-existing relationship with an Application Server. Any existing communication mechanism that is authenticated and provides confidentiality and integrity, such as HTTPS [RFC2818], is sufficient.

3. Message Encryption

An Application Server that has the key identifier, public key, group and format information can encrypt a message for the User Agent.

The Application Server generates a new DH or ECDH key pair in the same group as the value generated by the User Agent.

From the newly generated key pair, the Application Server performs a DH or ECDH computation with the public key provided by the User Agent to find the shared secret. The Application Server then generates 16 bytes of salt that is unique to the message. A random [RFC4086] salt is acceptable. These values are used to calculate the content encryption key as defined in Section 3.2 of [I-D.nottingham-http-encryption-encoding].

The Application Server then encrypts the payload. Header fields are populated with URL-safe base-64 encoded [RFC4648] values:

- the "keyid" from the User Agent is added to both the Encryption-Key and Encryption header fields;
- the salt is added to the "salt" parameter of the Encryption header field; and
- the public key for its DH or ECDH key pair is placed in the "dh" parameter of the Encryption-Key header field.

4. Message Decryption

A User Agent decrypts messages are decrypted as described in [I-D.nottingham-http-encryption-encoding]. The value of the "keyid" parameter is used to identify the correct key pair, if there is more than one possible value for the corresponding subscription.

5. Mandatory Group and Public Key Format

User Agents that enforce encryption MUST expose an elliptic curve Diffie-Hellman share on the P-256 curve [FIPS186]. Public keys, such as are encoded into the "dh" parameter, MUST be in the form of an uncompressed point as described in [X.692].

6. IANA Considerations

This document has no IANA actions.

7. Security Considerations

The security considerations of [I-D.nottingham-http-encryption-encoding] describe the limitations of the content encoding. In particular, any HTTP header fields are not protected by the content encoding scheme. A User Agent MUST consider HTTP header fields to have come from the Push Service. An application on the User Agent that uses information from header fields to alter their processing of a push message is exposed to a risk of attack by the Push Service.

The timing and length of communication cannot be hidden from the Push Service. While an outside observer might see individual messages intermixed with each other, the Push Service will see what Application Server is talking to which User Agent, and the subscription they are talking about. Additionally, the length of messages could be revealed unless the padding provided by the content encoding scheme is used to obscure length.

8. References

8.1. Normative References

[DH] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, V.IT-22 n.6 , June 1977.

[ECDH] SECG, "Elliptic Curve Cryptography", SEC 1 , 2000, <http://www.secg.org/>.

[FIPS186] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", NIST PUB 186-4 , July 2013.

[I-D.nottingham-http-encryption-encoding]
Nottingham, M. and M. Thomson, "Encrypted Content-Encoding for HTTP", draft-nottingham-http-encryption-encoding-00 (work in progress), March 2015.

[I-D.thomson-webpush-http2]
Thomson, M., "Generic Event Delivery Using HTTP Push", draft-thomson-webpush-http2-02 (work in progress), December 2014.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[X.692] ANSI, "Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62 , 1998.

8.2. Informative References

[API] Sullivan, B., Fullea, E., and M. van Ouwerkerk, "Web Push API", 2015, <https://w3c.github.io/push-api/>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC7230] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

Author’s Address

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com

WEBPUSH
Internet-Draft
Intended status: Standards Track
Expires: June 15, 2015

M. Thomson
Mozilla

December 12, 2014

Generic Event Delivery Using HTTP Push
draft-thomson-webpush-http2-02

Abstract

A simple protocol for the delivery of realtime events to clients is described. This scheme uses HTTP/2 server push.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on June 15, 2015.

Table of Contents

1. Introduction

Mobile computing devices are increasingly relied upon for a great many applications. Mobile devices typically have limited power reserves, so finding more efficient ways to serve application requirements is an important part of any mobile platform.

One significant contributor to power usage mobile devices is the radio. Radio communications consumes a significant portion of the energy budget on a wirelessly connected mobile device.

Many applications require continuous access to network communications so that real-time events - such as incoming calls or messages - can be conveyed (or "pushed") to the user in a timely fashion. Uncoordinated use of persistent connections or sessions from multiple applications can contribute to unnecessary use of the device radio, since each independent session independently incurs overheads. In particular, keep alive traffic used to ensure that middleboxes do not prematurely time out sessions, can result in significant waste.

Maintenance traffic tends to dominate over the long term, since events are relatively rare.

Consolidating all real-time events into a single session ensures more efficient use of network and radio resources. A single service consolidates all events, distributing those events to applications as they arrive. This requires just one session, avoiding duplicated overhead costs.

The W3C Web Push API [API] describes an API that enables the use of a consolidated push service from web applications. This expands on that work by describing a protocol that can be used to:

- request the delivery of a push message to a user agent,
- register a new user agent,
- create new push message delivery subscriptions, and
- monitor for new push messages.

This is intentionally split into these two categories because requesting the delivery of events is required for immediate use by the Web Push API. The registration, management and monitoring functions are currently fulfilled by proprietary protocols; these are adequate, but do not offer any of the advantages that standardization affords.

The monitoring function described in this document is intended to be replaceable, enabling the use of monitoring schemes that are better optimized for the network environment and the user agent. For instance, using notification systems like the 3GPP Short Message Service (SMS) [TS.3GPP.23.040] can take advantage of the native paging capabilities of a cellular network, avoiding the ongoing maintainence cost of a persistent TCP connection.

This document intentionally does not describe how a push service is discovered. Discovery of push services is left for future efforts, if it turns out to be necessary at all. User agents are expected to be configured with a URL for a push service.

Similarly, discovery of support for and negotiation of use of alternative monitoring schemes is left to documents that extend this basic protocol.

1.1. Conventions and Terminology

In cases where normative language needs to be emphasized, this document falls back on established shorthands for expressing interoperability requirements on implementations: the capitalized words "MUST", "MUST NOT", "SHOULD" and "MAY". The meaning of these is described in [RFC2119].

This document defines the following terms:

application: Both the sender and ultimate consumer of push messages. Many applications have components that are run on a user agent and other components that run on servers.

application server: The component of an application that runs on a server and requests the delivery of a push message.

push message: A message, sent from an application server to a user agent via a push service.

push service: A service that delivers push messages to user agents.

registration: A session that is established between the user agent and the push service. A registation has any number of associated subscriptions.

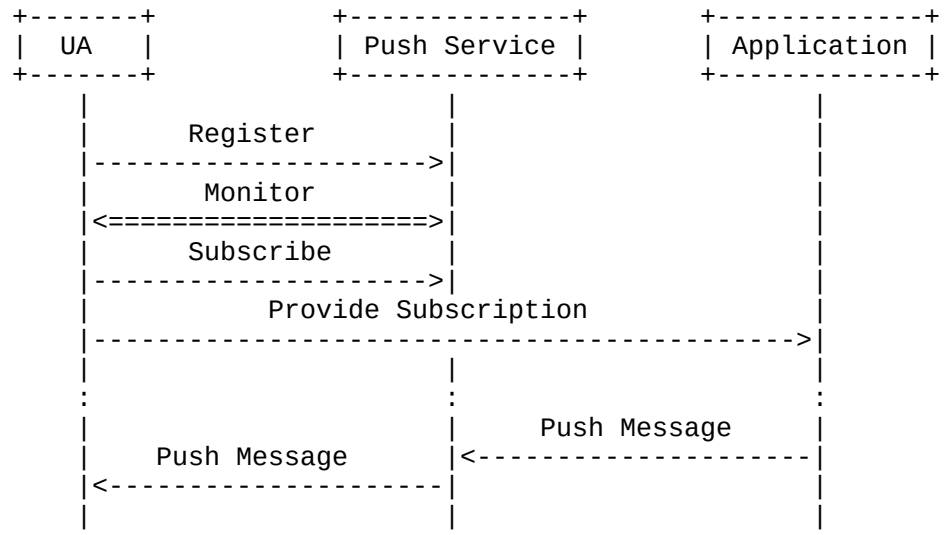
subscription: A message delivery context that is established between the user agent and the push service and shared with applications. All push messages are associated with a subscription.

user agent: A device and software that is the recipient of push messages.

Examples in this document use the HTTP/1.1 message format [RFC7230]. Many of the exchanges can be completed using HTTP/1.1, where HTTP/2 is necessary, the more verbose frame format from [I-D.ietf-httpbis-http2] is used.

2. Overview

A general model for push services includes three basic actors: a user agent, a push service, and an application (server).



At the very beginning of the process, the device creates a registration resource at the push service. The registration is the basis of all future interactions between the user agent and push service. The registration aggregates push messages that the user agent receives from all subscriptions.

The registration resource describes how the user agent is expected to monitor for incoming push messages. This document describes one such mechanism, though more efficient means of monitoring could be defined.

The registration resource also describes how the user agent might create a subscription. A new subscription is created by the user agent and then distributed by to an application server.

It is expected that a different subscription will be provided to each application; however, there are no inherent cardinality constraints in the protocol. Multiple subscriptions might be created for the same application, or multiple applications could use the same subscription. Note however that sharing subscriptions can have security and privacy implications.

Application servers use subscriptions to deliver push messages to devices, via the push service.

Both registrations and subscriptions have a limited lifetime. They can also be terminated by either push service or user agent at any time. User agents and application servers need to be prepared to handle changes in registrations and subscriptions; the protocol

described here supports any number of either resource concurrently with minimal overhead.

2.1. HTTP Resources

This protocol uses HTTP resources [RFC7230] and link relations [RFC5988]. The following resources are defined:

push service: This resource is used in Registration (Section 3). It is configured into user agents.

registration: A link relation of type "urn:ietf:params:push:reg" refers to a registration resource. This resource is used by a user agent in requesting the delivery of push messages. A process for monitoring for messages using this resource is described in Section 6.

subscribe: A link relation of type "urn:ietf:params:push:sub" refers to a resource where a user agent can create new subscriptions. Creating a subscription is described in Section 4.

subscription: A link relation of type "urn:ietf:params:push" refers to a subscription resource. Subscription resources are used to deliver push messages. An application server sends push messages (Section 5) and a user agent receives push messages (Section 6) using this resource.

3. Registration

A user agent that wishes to establish a new or replacement registration sends an HTTP POST request to its configured push service resource.

A request to create a registration contains no entity body. A future specification MAY define a format and semantics for the entity body on this request.

The push service creates a new registration and subscribe resource in response to this request. URIs for these resources are included in Link header fields in the response. The push server includes the "registration" link relation in a Location header field.

For example, the following request requests the creation of a new registration:

```
POST /register HTTP/1.1
Host: push.example.net
```

The following response might be generated in response to this request:

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:47 GMT
Link: </monitor/1G_GIPMorg_n-IrQvqZr6g>;
        rel="urn:ietf:params:push:reg"
Link: </subscribe/1G_GIPMorg_n-IrQvqZr6g>;
        rel="urn:ietf:params:push:sub"
Location: https://push.example.net/reg/1G_GIPMorg_n-IrQvqZr6g
Cache-Control: max-age=8640000, private
```

The push server SHOULD provide a URI for the associated "subscribe" resource and any known expiration information in response to requests to the "registration" resource.

4. Subscribing

A client sends a POST request to the "subscribe" resource to create a new subscription.

```
POST /subscribe/1G_GIPMorg_n-IrQvqZr6g HTTP/1.1
Host: push.example.net
```

A response with a 201 (Created) status code includes a URI for the subscription in the Location header field.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Link: </p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx>;
        rel="urn:ietf:params:push"
Location: https://push.example.net/p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx
Cache-Control: max-age:864000, private
```

5. Requesting Push Message Delivery

A push subscription is an HTTP resource [RFC7230].

An application server requests the delivery of a push message by sending an HTTP PUT request to this resource, including the push message in the body of the request.

```
PUT /p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx HTTP/1.1
Host: push.example.net
Content-Type: text/plain;charset=utf8
Content-Length: 15

Hello, World!
```

A simple 200 response is sufficient to indicate that the push message was accepted. This does not indicate that the message was delivered to the user agent.

```
HTTP/1.1 200 OK
Date: Thu, 11 Dec 2014 23:56:55 GMT
Cache-Control: max-age=600
```

A push service MAY generate a 413 (Payload Too Large) status code in response to PUT requests that include an entity body that is too large. Push services MUST NOT generate a 413 status code in responses to an entity body that is 4k (4096 bytes) or less in size.

A push service that does not store messages can stream the payload of push messages to the user agent. Flow control SHOULD be used to limit the state commitment for delivery of large messages.

6. Receiving Push Messages

A user agent requests the delivery of new push messages by making a GET request to the "registration" resource. The push service does not respond to this request, it instead uses HTTP/2 server push [I-D.ietf-httpbis-http2] to send the contents of push messages as they are sent by application servers.

Each push message is pushed in response to a synthesized GET request. The GET request is made to the same "subscription" URI that is used by the application server to send the message. The response body is the entity body from the most recent PUT request sent to the subscription resource.

The following example request is made over HTTP/2.

```
HEADERS      [stream 7] +END_STREAM +END_HEADERS
  :method        = GET
  :path          = /monitor/1G_GIPMorg_n-IrQvqZr6g
  :authority     = push.example.net
```

The push service permits the request to remain outstanding. When a push message is sent by an application server, a server push is associated with the initial request. The response includes the push message.

```
PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
  :method        = GET
  :path          = /p/LBhhw0OohO-Wl4Oi971UGsB7sdQGUibx
  :authority     = push.example.net

HEADERS      [stream 4] +END_HEADERS
  :status        = 200
  date           = Thu, 11 Dec 2014 23:56:55 GMT
  last-modified  = Thu, 11 Dec 2014 23:56:55 GMT
  cache-control  = private
  content-type   = text/plain;charset=utf8
  content-length = 15

DATA         [stream 4] +END_STREAM
  Hello World!\r\n
```

A user agent can request the contents of the "registration" resource immediately by including a Prefer header field [RFC7240] with a "wait" parameter set to "0". The push server SHOULD return a link reference to the "registration" resource and expiration information in response to this request. This request also triggers the delivery of all push messages that the push service has stored but not yet delivered.

A user agent can request the last push message for a "subscription" resource by sending GET requests to its URI. A 200 (OK) status response contains the last push message sent to the subscription. A 204 (No Content) status code indicates that no messages are presently available.

7. Operational Considerations

A push service is likely to have to maintain a very large number of open TCP connections. Effective management of those connections can depend on being able to move connections between server instances.

7.1. Load Management

A user agent MUST support the 307 (Temporary Redirect) status code [RFC7231], which can be used by a push service to redistribute load at the time that a new registration is requested.

A server that wishes to redistribute load can do so using the alternative services mechanisms that are part of HTTP/2 [I-D.ietf-httpbis-alt-svc]. Alternative services allows for redistribution of load whilst maintaining the same URIs for various resources. User agents can ensure a graceful transition by using the GOAWAY frame once it has established a replacement connection.

7.2. Push Message Expiration

Push services typically store messages for some time to allow for limited recovery from transient faults. If a push message is stored, but not delivered, the push service can indicate the probable duration of storage by including expiration information in the response to the push request.

A push service is not obligated to store messages indefinitely. If a user agent is not actively monitoring for push messages, those messages can be lost or overridden by newer messages on the same subscription.

Push messages that were stored and not delivered to a client are delivered when a client recommences monitoring. Stored push messages SHOULD include a Last-Modified header field (see Section 2.2 of [RFC7232]) indicating when delivery was requested by an application server.

A GET request to a "subscription" resource that has expired messages results in a 204 (No Content) status response, equivalent to if no push message were ever sent.

Push services might need to limit the size and number of stored push messages to avoid overloading. In addition to using the 413 (Payload Too Large) status code for too large push messages, a push service MAY expire push messages prior to any advertised expiration time.

7.3. Registration and Subscription Expiration

In some cases, it may be necessary to terminate registrations or subscriptions so that they can be refreshed.

A push service might choose to set a fixed expiration time. If a resource has a known expiration time, expiration information is included in responses to requests that create the resource, or in requests that retrieve a representation of the resource.

Expiration is indicated using either the Expires header field, or by setting a "max-age" parameter on a Cache-Control header field (see

[RFC7234]). The Cache-Control header field MUST also include the "private" directive.

A push service can invalidate a registration or subscription at any time. If a user agent has an outstanding request to the "registration" resource (see Section 6), this can be signaled by returning a 400-series response code, such as 410 (Gone). A push service uses server push to indicate that a subscription has expired; a pushed 400-series status code for the subscription resource signals the termination of a subscription.

A user agent can request that a registration or subscription be removed by sending a DELETE request to the corresponding URI.

7.4. Implications for Application Reliability

An application developer might find it tempting to create alternative mechanisms for message delivery in case the push service fails to deliver a critical message. Setting up a polling mechanism or a backup messaging channel in order to compensate for these shortcomings negates almost all of the advantages a push service provides.

Applications are encouraged to instead provide a means to detect situations where push messages were not delivered and recover gracefully. For instance, an application server might include a sequence number in push messages; a gap in the sequence can then be used to trigger some form of state resynchronization. For instance, the missing messages might be requested from the application server directly. Push service failures are expected to be rare, therefore performance optimization of any recovery mechanism might be unnecessary.

8. Security Considerations

This protocol MUST use HTTP over TLS [RFC2818]; this includes any communications between user agent and push service, plus communications between the application and the push service. Thus, all URIs use the "https" scheme. This provides confidentiality and integrity protection for registrations and push messages from external parties.

8.1. Confidentiality from Push Service Access

The protection afforded by TLS does not protect content from the push service. Without additional safeguards, a push service is able to see and modify the content of the messages.

Applications are able to provide additional confidentiality, integrity or authentication mechanisms within the push message itself. The application server sending the push message and the application on the user agent that receives it are frequently just different instances of the same application, so no standardized protocol is needed to establish a proper security context. The process of providing the application server with subscription information provides a convenient medium for key agreement.

The Web Push API codifies this practice by requiring that each push subscription created by the browser be bound to a browser generated encryption key. Pushed messages are authenticated and decrypted by the browser before delivery to applications. This scheme ensures that the push service is unable to examine the contents of push messages.

The public key for a subscription ensures that applications using that subscription can identify messages from unknown sources and discard them. This depends on the public key only being disclosed to entities that are authorized to send messages on the channel. The push server does not require access to this public key.

8.2. Privacy Considerations

Push message confidentiality does not ensure that the identity of who is communicating and when they are communicating is protected. However, the amount of information that is exposed can be limited.

Subscription URIs MUST NOT provide any basis to correlate communications for a given user agent. It MUST NOT be possible to correlate any two subscription URIs based solely on the content of the subscription URIs. This allows a user agent to control correlation across different applications, or over time.

In particular, user and device information MUST NOT be exposed through the subscription URI.

In addition, subscription URIs established by the same user agent MUST NOT include any information that allows them to be correlated with the associated registration or the user agent. The push service is the only entity that needs to be able to correlate subscriptions with a registration.

Note: This need not be perfect as long as the resulting anonymity set (see [RFC6973], Section 6.1.1) is sufficiently large. A subscription URI necessarily identifies a push service or a single server instance. It is also possible that traffic analysis could be used to correlate subscriptions.

A user agent MUST be able to create new registrations and subscriptions with new identifiers at any time.

8.3. Authorization

This protocol does not define how a push service establishes whether a user agent is permitted to create a registration or subscription, or whether push messages can be delivered to the user agent. A push service MAY choose to authorize request based on any HTTP-compatible authorization method available, of which there are numerous options. The authorization process and any associated credentials are expected to be configured in the user agent along with the URI for the "push service".

Authorization for sending push messages is managed using capability URLs (see [CAP-URI]). A capability URL grants access to a resource based solely on knowledge of the URL. Capability URLs are used for their "easy onward sharing" and "easy client API" properties. These make it possible to avoid relying on relationships between push services and application servers, with the protocols necessary to build and support those relationships.

A subscription URI therefore acts as a bearer token: knowledge of the URI implies authorization to send push messages. Subscription URIs MUST be extremely difficult to guess. Encoding a large amount of random entropy (at least 120 bits) in the path component ensures that it is difficult to successfully guess a valid subscription URI.

8.4. Denial of Service Considerations

Discarding unwanted messages at the user agent based on message authentication doesn’t protect against a denial of service attack on the user agent. Even a relatively small volume of push messages can cause battery-powered devices to exhaust power reserves. Limiting the number of entities with access to push channels limits the number of entities that can generate value push requests of the push server.

An application can limit where push messages can originate by limiting the distribution of subscription URIs to authorized entities. Ensuring that subscription URIs are hard to guess ensures that only applications servers that have been given a subscription URI can use it.

A malicious application with a valid subscription use the greater resources of a push service to mount a denial of service attack on a user agent. Push service SHOULD limit the rate at which push messages are sent to individual user agents. A push service or user

agent MAY terminate subscriptions (Section 7.3) that receives too many push messages.

End-to-end confidentiality mechanisms, such as those in [API], prevent an entity with a registration URI from learning the contents of push messages. In both cases, push messages that are not successfully authenticated will not be delivered by the API, but this can present a denial of service risk.

Conversely, a push service is also able to deny service to user agents. Intentional failure to deliver messages is difficult to distinguish from faults, which might occur due to transient network errors, interruptions in device availability, or genuine service outages.

8.5. Logging Risks

Server request logs can reveal registration and subscription URIs. Acquiring a registration URI permits the creation of new subscriptions, as well as potentially enabling the receipt of messages. Acquiring a subscription URI permits the sending of push messages. Logging could also reveal relationships between different subscription URIs for the same registration, or between different registrations for the same device.

Limitations on log retention and strong access control mechanisms can ensure that URIs are not learned by unauthorized entities.

9. IANA Considerations

This document registers three URNs for use in identifying link relation types. These are added to a new "Web Push Identifiers" registry according to the procedures in Section 4 of [RFC3553]; the corresponding "push" sub-namespace is entered in the "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry.

The "Web Push Identifiers" registry operates under the IETF Review policy [RFC5226].

Registry name: Web Push Identifiers

URN Prefix: urn:ietf:params:push

Specification: (this document)

Respository: [Editor/IANA note: please include a link to the final registry location.]

Index value: Values in this registry are URNs or URN prefixes that start with the prefix "urn:ietf:params:push". Each is registered independently.

New registrations in the "Web Push Identifiers" are encouraged to include the following information:

URN: A complete URN or URN prefix.

Description: A summary description.

Specification: A reference to a specification describing the semantics of the URN or URN prefix.

Contact: Email for the person or group making the registration.

Index value: As described in [RFC3553], URN prefixes that are registered include a description of how the URN is constructed. This is not applicable for specific URNs.

Three values are entered as the initial content of the "Web Push Identifiers" registry.

URN: urn:ietf:params:push

Description: This link relation type is used to identify a web push subscription.

Specification: (this document)

Contact: Martin Thomson (martin.thomson@gmail) or the Web Push WG (webpush@ietf.org)

URN: urn:ietf:params:push:reg

Description: This link relation type is used to identify a web push registration.

Specification: (this document)

Contact: Martin Thomson (martin.thomson@gmail) or the Web Push WG (webpush@ietf.org)

URN: urn:ietf:params:push:sub

Description: This link relation type is used to identify a resource that can be used to create new web push subscriptions.

Specification: (this document)

Contact: Martin Thomson (martin.thomson@gmail) or the Web Push WG (webpush@ietf.org)

10. Acknowledgements

Significant technical input to this document has been provided by Costin Manolache, Robert Sparks, Mark Nottingham, Matthew Kaufman and many others.

11. References

11.1. Normative References

[CAP-URI] Tennison, J., "Good Practices for Capability URLs", FPWD capability-urls, February 2014, <http://www.w3.org/TR/capability-urls/>.

[I-D.ietf-httpbis-alt-svc] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", draft-ietf-httpbis-alt-svc-01 (work in progress), April 2014.

[I-D.ietf-httpbis-http2] Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol version 2", draft-ietf-httpbis-http2-12 (work in progress), April 2014.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, June 2003.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.

[RFC7230] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

[RFC7231] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.

[RFC7232] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, June 2014.

[RFC7234] Fielding, R., Nottingham, M., and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, June 2014.

[RFC7240] Snell, J., "Prefer Header for HTTP", RFC 7240, June 2014.

11.2. Informative References

[API] Sullivan, B. and E. Fullea, "Web Push API", ED push-api, December 2014, <https://w3c.github.io/push-api/>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, July 2013.

[TS.3GPP.23.040] 3GPP, "Technical realization of the Short Message Service (SMS)", 3GPP TS 23.040 12.2.0, October 2014.

Author’s Address

Martin Thomson
Mozilla
331 E Evelyn Street
Mountain View, CA 94041
US

Email: martin.thomson@gmail.com