Web Security	C. Evans
Internet-Draft	C. Palmer
Expires: March 24, 2012	Google, Inc.
	September 21, 2011

Certificate Pinning Extension for HSTS
draft-evans-palmer-hsts-pinning-00

Abstract

This memo describes an extension to the HTTP Strict Transport Security specification allowing web host operators to instruct UAs to remember ("pin") hosts' cryptographic identities for a given period of time. During that time, UAs will require that the host present a certificate chain including at least one public key whose fingerprint matches one or more of the pinned fingerprints for that host. By effectively reducing the scope of authorities who can authenticate the domain during the lifetime of the pin, we hope pinning reduces the incidence of man-in-the-middle attacks due to compromised Certification Authorities and other authentication errors and attacks.

Status of this Memo

Copyright Notice

1. Introduction

We propose to extend the HSTS HTTP header to enable a web host to express to UAs which certificate(s) UAs may expect to be present in the host's certificate chain in future connections. We call this "certificate pinning". The Google Chrome/ium browser ships with a static set of pins, and individual users can extend the set of pins. Although effective, this does not scale. This proposal addresses the scale problem.

Deploying certificate pinning safely will require operational and organizational maturity due to the risk that HSTS Hosts may "brick" themselves by pinning to a certificate that becomes invalid. We discuss potential mitigations for those risks. We believe that, with care, host operators can greatly reduce the risk of MITM attacks and other false-authentication problems for their users without incurring undue risk.

This document extends the version of HSTS defined in [hsts-spec] and follows that document's notational and naming conventions.

This draft is being discussed on the WebSec Working Group mailing list, websec@ietf.org.

1.1. About Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

This document includes some pseudocode examples written in a Python-like language, to clarify UA behavior. The examples assume that a global data structure, hsts_metadata, exists and contains the HSTS metadata that the UA has accumulated over time. It is indexable by domain name and includes the usual HSTS parameters (maxAge, includesSubDomains) as well as the new HSTS parameter, pins, that this document introduces. It also assumes a hypothetical X.509 datatype, denoted with a variable named "certificate", that includes likely X.509 fields such as public_key (which would correspond to the SubjectPublicKeyInfo field in a real X.509 certificate).

There are also some working code examples using the Python and Go languages.

The examples are intended to be illustrative, not necessarily precise or using algorithms that a real, optimized UA would employ.

2. Server and Client Behavior

To set a pin, HSTS Hosts use a new STS extension directive (STS-d-ext) in their HSTS response header field: pins. To enable pin revocation (Section 2.3), hosts may also use the new breakv and breakc directives.

```
STS-d-ext-pin    =    "pins" OWS "=" OWS [fingerprints]
STS-d-ext-breakv =    "breakv" OWS "=" OWS fp-type "/" base64-digits
STS-d-ext-breakc =    "breakc" OWS "=" OWS base64-digits

fingerprints     =     fingerprint
                       / fingerprint "," fingerprints

fingerprint      =     fp-type "/" base64-digits

fp-type          =     "sha1"
                       / "sha256"
```

Here is an example response header field using the pins extension (folded for clarity):

```
Strict-Transport-Security: max-age=500; includeSubDomains;
    pins=sha1/4n972HfV354KP560yw4uqe/baXc=,
    sha1/IvGeLsbqzPxdI0b0wuj2xVTdXgc=
```

Here is an example response header field using both the pins and the breakv extensions (folded for clarity):

```
Strict-Transport-Security: max-age=500; includeSubDomains;
    pins=sha1/4n972HfV354KP560yw4uqe/baXc=,
    sha1/IvGeLsbqzPxdI0b0wuj2xVTdXgc=;
    breakv=sha1/jUQEXH7Q2Ly+Xn/yFWJxAHT3fDc=
```

The fingerprint is the SHA-1 (or SHA-256) hash of the raw SubjectPublicKeyInfo field of the certificate, encoded in base-64 for brevity. We pin public keys, rather than entire certificates, to enable operators to generate new certificates containing old public keys (see [why-fingerprint-key]). (Although host operators may do this, certification authorities already do. Additionally, when UAs check certificate chains, they do so by checking that each certificate is signed by its parent's public key, making the public key — not the certificate — the essential identifier.)
See Appendix Appendix A for an example program that generates public key fingerprints from SubjectPublicKeyInfo fields in certificates.
The breakv directive communicates to UAs a pin break verifier, and the breakc directive communicates the pin break code. Hosts SHOULD generate pin break codes and verifiers. When present, UAs MUST note pin break verifiers and honor pin break codes. See Section 2.3 for a discussion of verifiers and codes.

2.1. Noting and Validating Pins

Upon receipt of this header field, the UA will note the HSTS Host as a Known Pinned HSTS Host. When connecting to a Known Pinned HSTS Host, the UA will compare the public key fingerprint(s) in the Host's certificate chain to the pinned fingerprints, and will fail closed

unless at least one public key in the chain has a fingerprint matching one of the pinned fingerprints. (Following the HSTS specification, TLS errors for HSTS hosts must be hard, with no chance for the user to click through any warnings or errors. We treat fingerprint mismatch in the same way.)
Note that to validate pins, UAs must necessarily read the headers of a response. In case of mismatch, UAs SHOULD NOT read the response body as part of failing hard.
This pseudocode illustrates how UAs validate the certificate chains they receive from Known Pinned HSTS Hosts.

```
def chain_is_pinned_valid(chain, pins):
    for certificate in chain:
        for fingerprint in pins:
            if certificate.public_key.fingerprint == fingerprint:
                return True

    return False

# ...
if not chain_is_pinned_valid(request.tls_info.certificate_chain,
                             hsts_metadata[request.hostname].pins):
    request.fail()
# ...
```

The pin list appearing in an HSTS header MUST have at least one pin matching one of the public key fingerprints in the chain that was validated for the HTTPS connection. This defends against HTTP header injection attacks (see Section 3.4.1).
UAs MUST cache pins and pin break verifiers for Known Pinned HSTS Hosts, and MIGHT AS WELL do so in the same manner as other HSTS metadata. If the maxAge directive is present in the HSTS response header, the HSTS metadata — including fingerprints in the pins directive — expire at that time.

2.2. Interactions With Built-in HSTS Lists

UAs MAY choose to implement built-in certificate pins, alongside any built-in HSTS opt-in list. UAs MUST allow users to override a built-in pin list, including turning it off.
Hosts can update built-in pin lists by using this extension. Similarly, UAs can update their built-in pin lists with software updates. In either case, UAs MUST use the newest information — built-in or set via HSTS — when validating certificate chains for the host.

2.3. Un-pinning

Hosts can enable pin revocation for their previously-pinned key fingerprints by setting pin break verifiers using the breakv directive. Then, when hosts want to break pins, they set the pin break code in

their HSTS headers using the breakc directive. (This idea is due to Perrin in [pin-break-codes].)
Pin break codes are short random strings, kept secret until the host operator wants to break the pins. Pin break verifiers are simply hashes of the codes. Generating codes and verifiers, and verifying that codes match a previously set verifier, is trivial. See Figure 5.

```
def make_pin_break():
    code = os.urandom(16)
    verifier = hashlib.sha1(code).digest()
    return base64.b64encode(code), base64.b64encode(verifier)

def verify_code(code, verifier):
    c = base64.b64decode(code)
    v = hashlib.sha1(c).digest()
    return verifier == base64.b64encode(v)


if __name__ == "__main__":
    import sys

    if 1 == len(sys.argv):
        print make_pin_break()
    elif 3 == len(sys.argv):
        print verify_code(sys.argv[1], sys.argv[2])
```

Hosts can request that UAs forget pinned fingerprints by issuing a valid HSTS header containing the pin break code. UAs MUST forget all pinned fingerprints associated with the matching pin break verifier, and MUST NOT forget any pinned fingerprints not associated with that verifier.
In the event that a host sends an HSTS header containing a breakc that does not match a breakv the UA has previously noted, the UA MUST ignore that breakc and MUST process any pins or breakv directives as normal. This is so that hosts can break old pins but still successfully set new pins and verifiers in UAs that have not previously (or recently) noted the host.
Host operators SHOULD keep the pin break code secret, and SHOULD generate codes that are computationally infeasible to guess (such as by using their system's cryptographic random number generator; note that a 128-bit security level suffices).

2.4. Pinning Self-Signed Leaf Certificates

To the extent that UAs allow or enable hosts to authenticate themselves with self-signed end entity certificates, they MAY also allow hosts to pin the public keys in such certificates. The usability and security implications of this practice are outside the scope of this specification.

3. Security Considerations

3.1. Deployment Guidance

To recover from disasters of various types, as described below, we recommend that HSTS Hosts follow these guidelines.

*Operators SHOULD have a safety net: they should generate a backup key pair, get it signed by a different (root and/or intermediary) CA than their live certificate(s), store it safely offline, and set this backup pin in their pins directive.

  -Having a backup certificate was always a good idea anyway.

*It is most economical to have the backup certificate signed by a completely different signature chain than the live certificate, to maximize recoverability in the event of either root or intermediary signer compromise.

*Operators SHOULD periodically exercise their backup pin plan — an untested backup is no backup at all.

*Operators SHOULD have a diverse certificate portfolio. They should pin to a few different roots, owned by different companies if possible.

*Operators SHOULD start small. Operators SHOULD first deploy HSTS certificate pinning by setting a maxAge of minutes or a few hours, and gradually increase maxAge as they gain confidence in their operational capability.

3.2. Disasters Relating to Compromises of Certificates

3.2.1. The private key for the pinned leaf is stolen

If a leaf certificate is compromised, the host is likely to have experienced a complete compromise, in which case the problem is greater than certificates and pins. See Section 3.4.2.

3.2.2. The root or intermediary CA is compromised

This disaster will affect many hosts (HSTS Hosts and other), and will likely require a client software update (e.g. to revoke the signing CA and/or the false certificates it issued).
If the operator has a backup pin whose signature chain is still valid, they should deploy it. In this case, the host need not even degrade from Known Pinned to Known.

3.3. Disasters Relating to Certificate Mismanagement

3.3.1. The leaf certificate expires

Operators should deploy their backup pin.
Note that when evaluating a pinned certificate, the UA MUST un-pin the fingerprint if the certificate has expired. If a pin list becomes empty, the UA downgrades the host from Known Pinned HSTS Host to Known HSTS Host. The usual HTTPS validation procedure now applies.
Operators should get any CA to sign a new cert with updated expiry, based on the existing, unchanged public key.

*And/or, operators should deploy their backup pin and/or have a CA sign an all-new key.

*Operators should continue to set pins in their HSTS header, and UAs will upgrade from Known HSTS Host to Known Pinned HSTS Host when the fingerprint(s) refer(s) to valid certificate(s) again.

3.3.2. The leaf certificate is lost

Operators should deploy their backup pin. Alternately, if they pinned to a root or intermediary signer, they should get a new leaf certificate signed by one of those signers.
Operators SHOULD attempt to get the certificate revoked by whatever means available (extant revocation mechanisms like CRL or OCSP, blacklisting in the UA, or future revocation mechanisms).

*We know that extant revocation mechanisms are unreliable. Operators SHOULD NOT not depend on them.

3.3.3. The CA is extorting the operator approaching renewal/expiry time

If the backup pin chains to a different signer, the operator should deploy it. (They should then get a new backup pin.)
The time running up to renewal can be used to serve additional HSTS public key hashes, pinning to new root CAs.

*Hosts can also disable pinning altogether as described above.

If the host is pinned to leaves or its own intermediary, operators can simply get a different root CA to sign the existing public key.
If the operator fails to get new certs in time, and the host is pinned only to the one root CA, the solution is simple; see Section 3.3.1.

3.4. Disasters Relating to Vulnerabilities in the Known HSTS Host

3.4.1. The host is vulnerable to HTTP header injection

Note that header injection vulnerabilities are in general more severe than merely disabling pinning for individual users.
The attacker could set additional pins for certificates he controls, or pin break verifiers for codes he controls, allowing him to undetectably MITM clients. When or if the client is outside the scope of the attacker's MITM attack, the result is DoS.
The attacker could disable HSTS and pins.

3.4.2. The host suffers full server-side compromise

After setting up a new host, operators should deploy the backup pin. Alternately, if the host is pinned to a root or intermediary signer, the operator should get a new leaf certificate signed by one of those signers.
Operators SHOULD attempt to get the certificate containing the compromised private key revoked by whatever means available (extant revocation mechanisms like CRL or OCSP, blacklisting in the UA, or future revocation mechanisms).

> *We know that extant revocation mechanisms are unreliable. Do not depend on them.

4. Usability Considerations

When pinning works to detect impostor Known Pinned HSTS Hosts, users will experience denial of service. UAs SHOULD explain the reason why. If it happens that true positives (actual attacks) outnumber false positives (hosts bricking themselves by accident), the feature will gain a positive reputation. Note that pinning has started life with a good reputation because it provoked the discovery of the DigiNotar CA compromise. (When DigiNotar signed a certificate for *.google.com in August 2011, Chrome users discovered the attack due to the pre-loaded pins for Google domains.)
We believe that, in general, DoS is a better failure mode than user account/session compromise or other result of TLS compromise.
UAs MUST have a way for users to clear current pins that were set by HSTS. UAs SHOULD have a way for users to query the current state of Known (Pinned) HSTS Hosts.

5. Economic Considerations

If pinning becomes common, host operators might become incentivized to choose CAs that get compromised less often, or respond better to compromise. This will require information to flow into the market, and for people to interpret no news post-compromise as bad news. Pinning itself will provide some of that information, as will sources like UA

vendor communications, the EFF SSL Observatory, the Qualys SSL survey, etc.
The disaster recovery plans described above all incur new costs for host operators, and increase the size of the certificate market. Arguably, well-run hosts had already absorbed these costs because (e.g.) backup certificates from different CAs were necessary disaster recovery mechanisms even before certificate pinning. Small sites — which although small might still need to provide good security — may not be able to afford the disaster recovery mechanisms we recommend. (The cost of the backup certificate is not the issue; it is more the operational costs in safely storing the backup and testing that it works.) Thus, low-risk pinning may be available only to large sites; small sites may have to choose no pinning or potentially bricking their host (up to the maxAge window). This is not worse than the status quo.

6. Ideas

6.1. Requiring Backup Pins

Because bricking risk mitigation requires a backup pin, UAs could require that the pins directive have at least two fingerprints, at least one of which does not match any of the public keys in any of the certificates in the chain. (This idea due to Tom Sepez.)

6.2. Prepopulating Pin Lists

HSTS-based certificate pinning, unlike built-in pinning, suffers from the bootstrap problem. To work around this, we could pre-populate a built-in pin list with public keys as observed in the wild by one or more global observers, such as Googlebot, the EFF SSL Observatory, Convergence notaries, and so on.
One problem with this approach is that it does not involve host operators. It is best to get operator consent before signing them up for a potentially risky new protocol such as this. Therefore we leave this idea for work (including third-party UA extensions).

6.3. Tools to Assist Creation of Header

It would be good to provide tools that read X.509 certificate chains and generate example HSTS headers that operators can easily add to their webs erver configurations.

6.4. Pinning Subresources

Many hosts have pages that load subresources from domains not under the control, or under only partial control, of the main host's operators. For example, popular hosts often use CDNs, and CDN customers may have only limited, if any, ability to influence the configuration of the CDN's servers. (This long-standing problem is independent of certificate pinning.)

To a limited extent, the includeSubDomains HSTS directive can address this: if the CDN host has a name that is a subdomain of the main host (e.g. assets-from-cdn.example.com points to CDN-owned servers), and if the main host's operators can guaranteeably keep up-to-date with the CDN's server certificate fingerprints — perhaps as part of example.com's contract with the CDN — then the problem may be solved. CDNs SHOULD also use certificate pinning independently of any of their customers.

Although one can imagine an extension to this specification allowing the main resource to set pins for subresources in other domains, it is complex and fragile both from technical and business perspectives. The UA would have to accept those pins for the subresource domains ONLY when loading resources from the subdomains as part of a page load of the main host. The independence of the two domains' operations teams would still pose synchronization problems, and potentially increase the bricking risk.

Therefore, except in simple cases, this document leaves the cross-domain subresource problem to future work. Operational experience with HSTS-based certificate pinning should guide the development of a plan to handle the problem.

6.5. Pinning Without Requiring HTTPS

Some host operators would like to take advantage of certificate pinning without requiring HTTPS, but having clients require pins in the event that they do connect to the host with HTTPS. As specified above, the current HSTS-based mechanism does not allow for this: clients that receive the pins directive via HSTS will also therefore require HTTPS — that is the purpose of HSTS after all. To have an additional directive, e.g. mode=optional, would not work because older clients that support HSTS but not the mode extension would effectively require HTTPS.

Alternatives include (a) putting the pins directive in a new header instead of extending HSTS; and (b) some kind of hack like setting maxAge=0 and having an additional directive to keep the pins alive (e.g. pinMaxAge). These alternatives seem ugly to us and we welcome suggestions for a better way to support this deployment scenario.

7. Acknowledgements

Thanks to Jeff Hodges, Adam Langley, Nicolas Lidzborski, SM, and Yoav Nir for suggestions and edits that clarified the text. Trevor Perrin for providing the pin break codes mechanism. Adam Langley provided the SPKI fingerprint generation code.

8. What's Changed

This is the first draft of this proposal submitted as an official Internet Draft.

9. References

[hsts-spec]	Hodges, J., Jackson, C. and A. Barth, "HTTP Strict Transport Security (HSTS)", August 2011.
[why-fingerprint-key]	Langley, A., "Public Key Pinning", May 2011.
[pin-break-codes]	Perrin, T., "Self-Asserted Key Pinning", September 2011.
[rfc-2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997.

Appendix A. Fingerprint Generation

This Go program generates public key fingerprints, suitable for use in pinning, from PEM-encoded certificates.

```
package main

import (
	"io/ioutil"
	"os"
	"crypto/sha1"
	"crypto/x509"
	"encoding/base64"
	"encoding/pem"
	"fmt"
)

func main() {
	if len(os.Args) < 2 {
		fmt.Printf("Usage: %s PEM-filename\n", os.Args[0])
		os.Exit(1)
	}
	pemBytes, err := ioutil.ReadFile(os.Args[1])
	if err != nil {
		panic(err.String())
	}
	block, _ := pem.Decode(pemBytes)
	if block == nil {
		panic("No PEM structure found")
	}
	derBytes := block.Bytes
	certs, err := x509.ParseCertificates(derBytes)
	if err != nil {
		panic(err.String())
	}
	cert := certs[0]
	h := sha1.New()
	h.Write(cert.RawSubjectPublicKeyInfo)
	digest := h.Sum()

	fmt.Printf("Hex: %x\nBase64: %s\n", digest,
		base64.StdEncoding.EncodeToString(digest))
}
```

Authors' Addresses

Chris Evans Evans Google, Inc. 1600 Amphitheater Pkwy Mountain View, CA 94043 US EMail: cevans@google.com

Chris Palmer Palmer Google, Inc. 1600 Amphitheater Pkwy Mountain View, CA 94043 US EMail: palmer@google.com

Table of Contents