NAME

X509_VERIFY_PARAM_set_flags, X509_VERIFY_PARAM_clear_flags, X509_VERIFY_PARAM_get_flags, X509_VERIFY_PARAM_set_purpose, X509_VERIFY_PARAM_get_inh_flags, X509_VERIFY_PARAM_set_inh_flags, X509_VERIFY_PARAM_set_trust, X509_VERIFY_PARAM_set_depth, X509_VERIFY_PARAM_get_depth, X509_VERIFY_PARAM_set_auth_level, X509_VERIFY_PARAM_get_auth_level, X509_VERIFY_PARAM_set_time, X509_VERIFY_PARAM_get_time, X509_VERIFY_PARAM_add0_policy, X509_VERIFY_PARAM_set1_policies, X509_VERIFY_PARAM_get0_host, X509_VERIFY_PARAM_set1_host, X509_VERIFY_PARAM_add1_host, X509_VERIFY_PARAM_set_hostflags, X509_VERIFY_PARAM_get_hostflags, X509_VERIFY_PARAM_get0_peername, X509_VERIFY_PARAM_get0_email, X509_VERIFY_PARAM_set1_email, X509_VERIFY_PARAM_set1_ip, X509_VERIFY_PARAM_get1_ip_asc, X509_VERIFY_PARAM_set1_ip_asc - X509 verification parameters

SYNOPSIS

```
#include <openssl/x509_vfy.h>

int X509_VERIFY_PARAM_set_flags(X509_VERIFY_PARAM *param,
                unsigned long flags);
int X509_VERIFY_PARAM_clear_flags(X509_VERIFY_PARAM *param,
                unsigned long flags);
unsigned long X509_VERIFY_PARAM_get_flags(const X509_VERIFY_PARAM *param);

int X509_VERIFY_PARAM_set_inh_flags(X509_VERIFY_PARAM *param,
                uint32_t flags);
uint32_t X509_VERIFY_PARAM_get_inh_flags(const X509_VERIFY_PARAM *param);

int X509_VERIFY_PARAM_set_purpose(X509_VERIFY_PARAM *param, int purpose);
int X509_VERIFY_PARAM_set_trust(X509_VERIFY_PARAM *param, int trust);

void X509_VERIFY_PARAM_set_time(X509_VERIFY_PARAM *param, time_t t);
time_t X509_VERIFY_PARAM_get_time(const X509_VERIFY_PARAM *param);

int X509_VERIFY_PARAM_add0_policy(X509_VERIFY_PARAM *param,
                ASN1_OBJECT *policy);
int X509_VERIFY_PARAM_set1_policies(X509_VERIFY_PARAM *param,
                STACK_OF(ASN1_OBJECT) *policies);
```

```
    void X509_VERIFY_PARAM_set_depth(X509_VERIFY_PARAM *param, int depth);
    int X509_VERIFY_PARAM_get_depth(const X509_VERIFY_PARAM *param);

    void X509_VERIFY_PARAM_set_auth_level(X509_VERIFY_PARAM *param,
                        int auth_level);
    int X509_VERIFY_PARAM_get_auth_level(const X509_VERIFY_PARAM *param);

    char *X509_VERIFY_PARAM_get0_host(X509_VERIFY_PARAM *param, int n);
    int X509_VERIFY_PARAM_set1_host(X509_VERIFY_PARAM *param,
                    const char *name, size_t namelen);
    int X509_VERIFY_PARAM_add1_host(X509_VERIFY_PARAM *param,
                    const char *name, size_t namelen);
    void X509_VERIFY_PARAM_set_hostflags(X509_VERIFY_PARAM *param,
                        unsigned int flags);
    unsigned int X509_VERIFY_PARAM_get_hostflags(const X509_VERIFY_PARAM *param);
    char *X509_VERIFY_PARAM_get0_peername(const X509_VERIFY_PARAM *param);
    char *X509_VERIFY_PARAM_get0_email(X509_VERIFY_PARAM *param);
    int X509_VERIFY_PARAM_set1_email(X509_VERIFY_PARAM *param,
                    const char *email, size_t emaillen);
    char *X509_VERIFY_PARAM_get1_ip_asc(X509_VERIFY_PARAM *param);
    int X509_VERIFY_PARAM_set1_ip(X509_VERIFY_PARAM *param,
                    const unsigned char *ip, size_t iplen);
    int X509_VERIFY_PARAM_set1_ip_asc(X509_VERIFY_PARAM *param, const char *ipasc);
```

DESCRIPTION

These functions manipulate the **X509_VERIFY_PARAM** structure associated with a certificate verification operation.

The **X509_VERIFY_PARAM_set_flags()** function sets the flags in **param** by oring it with **flags**. See "VERIFICATION FLAGS" for a complete description of values the **flags** parameter can take.

X509_VERIFY_PARAM_get_flags() returns the flags in **param**.

X509_VERIFY_PARAM_get_inh_flags() returns the inheritance flags in **param** which specifies how verification flags are copied from one structure to another. **X509_VERIFY_PARAM_set_inh_flags()** sets the inheritance flags. See the **INHERITANCE FLAGS** section for a description of these bits.

X509_VERIFY_PARAM_clear_flags() clears the flags **flags** in **param**.

X509_VERIFY_PARAM_set_purpose() sets the verification purpose in **param** to **purpose**. This

determines the acceptable purpose of the certificate chain, for example
X509_PURPOSE_SSL_CLIENT. The purpose requirement is cleared if **purpose** is 0.

X509_VERIFY_PARAM_set_trust() sets the trust setting in **param** to **trust**.

X509_VERIFY_PARAM_set_time() sets the verification time in **param** to **t**. Normally the current time is used.

X509_VERIFY_PARAM_add0_policy() adds **policy** to the acceptable policy set. Contrary to preexisting documentation of this function it does not enable policy checking.

X509_VERIFY_PARAM_set1_policies() enables policy checking (it is disabled by default) and sets the acceptable policy set to **policies**. Any existing policy set is cleared. The **policies** parameter can be **NULL** to clear an existing policy set.

X509_VERIFY_PARAM_set_depth() sets the maximum verification depth to **depth**. That is the maximum number of intermediate CA certificates that can appear in a chain. A maximal depth chain contains 2 more certificates than the limit, since neither the end-entity certificate nor the trust-anchor count against this limit. Thus a **depth** limit of 0 only allows the end-entity certificate to be signed directly by the trust anchor, while with a **depth** limit of 1 there can be one intermediate CA certificate between the trust anchor and the end-entity certificate.

X509_VERIFY_PARAM_set_auth_level() sets the authentication security level to **auth_level**. The authentication security level determines the acceptable signature and public key strength when verifying certificate chains. For a certificate chain to validate, the public keys of all the certificates must meet the specified security level. The signature algorithm security level is not enforced for the chain's *trust anchor* certificate, which is either directly trusted or validated by means other than its signature. See **SSL_CTX_set_security_level**(3) for the definitions of the available levels. The default security level is -1, or "not set". At security level 0 or lower all algorithms are acceptable. Security level 1 requires at least 80-bit-equivalent security and is broadly interoperable, though it will, for example, reject MD5 signatures or RSA keys shorter than 1024 bits.

X509_VERIFY_PARAM_get0_host() returns the **n**th expected DNS hostname that has been set using **X509_VERIFY_PARAM_set1_host()** or **X509_VERIFY_PARAM_add1_host()**. To obtain all names start with **n** = 0 and increment **n** as long as no NULL pointer is returned.

X509_VERIFY_PARAM_set1_host() sets the expected DNS hostname to **name** clearing any previously specified hostname. If **name** is NULL, or empty the list of hostnames is cleared, and name checks are not performed on the peer certificate. If **name** is NUL-terminated, **namelen** may be zero, otherwise **namelen** must be set to the length of **name**.

When a hostname is specified, certificate verification automatically invokes **X509_check_host**(3) with flags equal to the **flags** argument given to **X509_VERIFY_PARAM_set_hostflags()** (default zero). Applications are strongly advised to use this interface in preference to explicitly calling **X509_check_host**(3), hostname checks may be out of scope with the **DANE-EE**(3) certificate usage, and the internal check will be suppressed as appropriate when DANE verification is enabled.

When the subject CommonName will not be ignored, whether as a result of the **X509_CHECK_FLAG_ALWAYS_CHECK_SUBJECT** host flag, or because no DNS subject alternative names are present in the certificate, any DNS name constraints in issuer certificates apply to the subject CommonName as well as the subject alternative name extension.

When the subject CommonName will be ignored, whether as a result of the **X509_CHECK_FLAG_NEVER_CHECK_SUBJECT** host flag, or because some DNS subject alternative names are present in the certificate, DNS name constraints in issuer certificates will not be applied to the subject DN. As described in **X509_check_host**(3) the **X509_CHECK_FLAG_NEVER_CHECK_SUBJECT** flag takes precedence over the **X509_CHECK_FLAG_ALWAYS_CHECK_SUBJECT** flag.

X509_VERIFY_PARAM_get_hostflags() returns any host flags previously set via a call to **X509_VERIFY_PARAM_set_hostflags()**.

X509_VERIFY_PARAM_add1_host() adds **name** as an additional reference identifier that can match the peer's certificate. Any previous names set via **X509_VERIFY_PARAM_set1_host()** or **X509_VERIFY_PARAM_add1_host()** are retained, no change is made if **name** is NULL or empty. When multiple names are configured, the peer is considered verified when any name matches.

X509_VERIFY_PARAM_get0_peername() returns the DNS hostname or subject CommonName from the peer certificate that matched one of the reference identifiers. When wildcard matching is not disabled, or when a reference identifier specifies a parent domain (starts with ".") rather than a hostname, the peer name may be a wildcard name or a sub-domain of the reference identifier respectively. The return string is allocated by the library and is no longer valid once the associated **param** argument is freed. Applications must not free the return value.

X509_VERIFY_PARAM_get0_email() returns the expected RFC822 email address.

X509_VERIFY_PARAM_set1_email() sets the expected RFC822 email address to **email**. If **email** is NUL-terminated, **emaillen** may be zero, otherwise **emaillen** must be set to the length of **email**. When an email address is specified, certificate verification automatically invokes **X509_check_email**(3).

X509_VERIFY_PARAM_get1_ip_asc() returns the expected IP address as a string. The caller is

responsible for freeing it.

X509_VERIFY_PARAM_set1_ip() sets the expected IP address to **ip**. The **ip** argument is in binary format, in network byte-order and **iplen** must be set to 4 for IPv4 and 16 for IPv6. When an IP address is specified, certificate verification automatically invokes **X509_check_ip**(3).

X509_VERIFY_PARAM_set1_ip_asc() sets the expected IP address to **ipasc**. The **ipasc** argument is a NUL-terminal ASCII string: dotted decimal quad for IPv4 and colon-separated hexadecimal for IPv6. The condensed "::" notation is supported for IPv6 addresses.

RETURN VALUES

X509_VERIFY_PARAM_set_flags(), **X509_VERIFY_PARAM_clear_flags()**, **X509_VERIFY_PARAM_set_inh_flags()**, **X509_VERIFY_PARAM_set_purpose()**, **X509_VERIFY_PARAM_set_trust()**, **X509_VERIFY_PARAM_add0_policy()** **X509_VERIFY_PARAM_set1_policies()**, **X509_VERIFY_PARAM_set1_host()**, **X509_VERIFY_PARAM_add1_host()**, **X509_VERIFY_PARAM_set1_email()**, **X509_VERIFY_PARAM_set1_ip()** and **X509_VERIFY_PARAM_set1_ip_asc()** return 1 for success and 0 for failure.

X509_VERIFY_PARAM_get0_host(), **X509_VERIFY_PARAM_get0_email()**, and **X509_VERIFY_PARAM_get1_ip_asc()**, return the string pointer specified above or NULL if the respective value has not been set or on error.

X509_VERIFY_PARAM_get_flags() returns the current verification flags.

X509_VERIFY_PARAM_get_hostflags() returns any current host flags.

X509_VERIFY_PARAM_get_inh_flags() returns the current inheritance flags.

X509_VERIFY_PARAM_set_time() and **X509_VERIFY_PARAM_set_depth()** do not return values.

X509_VERIFY_PARAM_get_depth() returns the current verification depth.

X509_VERIFY_PARAM_get_auth_level() returns the current authentication security level.

VERIFICATION FLAGS

The verification flags consists of zero or more of the following flags ored together.

X509_V_FLAG_CRL_CHECK enables CRL checking for the certificate chain leaf certificate. An error occurs if a suitable CRL cannot be found.

X509_V_FLAG_CRL_CHECK_ALL enables CRL checking for the entire certificate chain.

X509_V_FLAG_IGNORE_CRITICAL disables critical extension checking. By default any unhandled critical extensions in certificates or (if checked) CRLs result in a fatal error. If this flag is set unhandled critical extensions are ignored. **WARNING** setting this option for anything other than debugging purposes can be a security risk. Finer control over which extensions are supported can be performed in the verification callback.

The **X509_V_FLAG_X509_STRICT** flag disables workarounds for some broken certificates and makes the verification strictly apply **X509** rules.

X509_V_FLAG_ALLOW_PROXY_CERTS enables proxy certificate verification.

X509_V_FLAG_POLICY_CHECK enables certificate policy checking, by default no policy checking is performed. Additional information is sent to the verification callback relating to policy checking.

X509_V_FLAG_EXPLICIT_POLICY, **X509_V_FLAG_INHIBIT_ANY** and **X509_V_FLAG_INHIBIT_MAP** set the **require explicit policy**, **inhibit any policy** and **inhibit policy mapping** flags respectively as defined in **RFC3280**. Policy checking is automatically enabled if any of these flags are set.

If **X509_V_FLAG_NOTIFY_POLICY** is set and the policy checking is successful a special status code is set to the verification callback. This permits it to examine the valid policy tree and perform additional checks or simply log it for debugging purposes.

By default some additional features such as indirect CRLs and CRLs signed by different keys are disabled. If **X509_V_FLAG_EXTENDED_CRL_SUPPORT** is set they are enabled.

If **X509_V_FLAG_USE_DELTAS** is set delta CRLs (if present) are used to determine certificate status. If not set deltas are ignored.

X509_V_FLAG_CHECK_SS_SIGNATURE requests checking the signature of the last certificate in a chain if the certificate is supposedly self-signed. This is prohibited and will result in an error if it is a non-conforming CA certificate with key usage restrictions not including the *keyCertSign* bit. By default this check is disabled because it doesn't add any additional security but in some cases applications might want to check the signature anyway. A side effect of not checking the self-signature of such a certificate is that disabled or unsupported message digests used for the signature are not treated as fatal errors.

When **X509_V_FLAG_TRUSTED_FIRST** is set, which is always the case since OpenSSL 1.1.0,

construction of the certificate chain in **X509_verify_cert**(3) searches the trust store for issuer certificates before searching the provided untrusted certificates. Local issuer certificates are often more likely to satisfy local security requirements and lead to a locally trusted root. This is especially important when some certificates in the trust store have explicit trust settings (see "TRUST SETTINGS" in **openssl-x509**(1)).

The **X509_V_FLAG_NO_ALT_CHAINS** flag could have been used before OpenSSL 1.1.0 to suppress checking for alternative chains. By default, unless **X509_V_FLAG_TRUSTED_FIRST** is set, when building a certificate chain, if the first certificate chain found is not trusted, then OpenSSL will attempt to replace untrusted certificates supplied by the peer with certificates from the trust store to see if an alternative chain can be found that is trusted. As of OpenSSL 1.1.0, with **X509_V_FLAG_TRUSTED_FIRST** always set, this option has no effect.

The **X509_V_FLAG_PARTIAL_CHAIN** flag causes non-self-signed certificates in the trust store to be treated as trust anchors, in the same way as self-signed root CA certificates. This makes it possible to trust self-issued certificates as well as certificates issued by an intermediate CA without having to trust their ancestor root CA. With OpenSSL 1.1.0 and later and **X509_V_FLAG_PARTIAL_CHAIN** set, chain construction stops as soon as the first certificate contained in the trust store is added to the chain, whether that certificate is a self-signed "root" certificate or a not self-signed "intermediate" or self-issued certificate. Thus, when an intermediate certificate is found in the trust store, the verified chain passed to callbacks may be shorter than it otherwise would be without the **X509_V_FLAG_PARTIAL_CHAIN** flag.

The **X509_V_FLAG_NO_CHECK_TIME** flag suppresses checking the validity period of certificates and CRLs against the current time. If **X509_VERIFY_PARAM_set_time()** is used to specify a verification time, the check is not suppressed.

INHERITANCE FLAGS

These flags specify how parameters are "inherited" from one structure to another.

If **X509_VP_FLAG_ONCE** is set then the current setting is zeroed after the next call.

If **X509_VP_FLAG_LOCKED** is set then no values are copied. This overrides all of the following flags.

If **X509_VP_FLAG_DEFAULT** is set then anything set in the source is copied to the destination. Effectively the values in "to" become default values which will be used only if nothing new is set in "from". This is the default.

If **X509_VP_FLAG_OVERWRITE** is set then all value are copied across whether they are set or not.

Flags is still Ored though.

If **X509_VP_FLAG_RESET_FLAGS** is set then the flags value is copied instead of ORed.

NOTES

The above functions should be used to manipulate verification parameters instead of functions which work in specific structures such as **X509_STORE_CTX_set_flags**() which are likely to be deprecated in a future release.

BUGS

Delta CRL checking is currently primitive. Only a single delta can be used and (partly due to limitations of **X509_STORE**) constructed CRLs are not maintained.

If CRLs checking is enable CRLs are expected to be available in the corresponding **X509_STORE** structure. No attempt is made to download CRLs from the CRL distribution points extension.

EXAMPLES

Enable CRL checking when performing certificate verification during SSL connections associated with an **SSL_CTX** structure **ctx**:

```
X509_VERIFY_PARAM *param;

param = X509_VERIFY_PARAM_new();
X509_VERIFY_PARAM_set_flags(param, X509_V_FLAG_CRL_CHECK);
SSL_CTX_set1_param(ctx, param);
X509_VERIFY_PARAM_free(param);
```

SEE ALSO

X509_verify_cert(3), **X509_check_host**(3), **X509_check_email**(3), **X509_check_ip**(3), **openssl-x509**(1)

HISTORY

The **X509_V_FLAG_NO_ALT_CHAINS** flag was added in OpenSSL 1.1.0. The flag **X509_V_FLAG_CB_ISSUER_CHECK** was deprecated in OpenSSL 1.1.0 and has no effect.

The **X509_VERIFY_PARAM_get_hostflags**() function was added in OpenSSL 1.1.0i.

The **X509_VERIFY_PARAM_get0_host**(), **X509_VERIFY_PARAM_get0_email**(), and **X509_VERIFY_PARAM_get1_ip_asc**() functions were added in OpenSSL 3.0.

The function **X509_VERIFY_PARAM_add0_policy**() was historically documented as enabling policy checking however the implementation has never done this. The documentation was changed to align with the implementation.

COPYRIGHT

Copyright 2009-2023 The OpenSSL Project Authors. All Rights Reserved.

Licensed under the Apache License 2.0 (the "License"). You may not use this file except in compliance with the License. You can obtain a copy in the file LICENSE in the source distribution or at <https://www.openssl.org/source/license.html>.