```
************************************************************
   57931 Mon May  5 11:11:31 2014
new/usr/src/uts/common/io/mac/mac_protect.c
4788 mac shouldn't abuse ddi_get_time(9f)
************************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

#include <sys/strsun.h>
#include <sys/sdt.h>
#include <sys/mac.h>
#include <sys/mac_impl.h>
#include <sys/mac_client_impl.h>
#include <sys/mac_client_priv.h>
#include <sys/ethernet.h>
#include <sys/vlan.h>
#include <sys/dlpi.h>
#include <sys/avl.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/arp.h>
#include <netinet/arp.h>
#include <netinet/udp.h>
#include <netinet/dhcp.h>
#include <netinet/dhcp6.h>

/*
 * Implementation overview for DHCP address detection
 *
 * The purpose of DHCP address detection is to relieve the user of having to
 * manually configure static IP addresses when ip-nospoof protection is turned
 * on. To achieve this, the mac layer needs to intercept DHCP packets to
 * determine the assigned IP addresses.
 *
 * A DHCP handshake between client and server typically requires at least
 * 4 messages:
 *
 * 1. DISCOVER - client attempts to locate DHCP servers via a
 *               broadcast message to its subnet.
 * 2. OFFER    - server responds to client with an IP address and
```

```
 *               other parameters.
 * 3. REQUEST  - client requests the offered address.
 * 4. ACK      - server verifies that the requested address matches
 *               the one it offered.
 *
 * DHCPv6 behaves pretty much the same way aside from different message names.
 *
 * Address information is embedded in either the OFFER or REQUEST message.
 * We chose to intercept REQUEST because this is at the last part of the
 * handshake and it indicates that the client intends to keep the address.
 * Intercepting OFFERs is unreliable because the client may receive multiple
 * offers from different servers, and we can't tell which address the client
 * will keep.
 *
 * Each DHCP message has a transaction ID. We use this transaction ID to match
 * REQUESTs with ACKs received from servers.
 *
 * For IPv4, the process to acquire a DHCP-assigned address is as follows:
 *
 * 1. Client sends REQUEST. a new dhcpv4_txn_t object is created and inserted
 *    in the the mci_v4_pending_txn table (keyed by xid). This object represents
 *    a new transaction. It contains the xid, the client ID and requested IP
 *    address.
 *
 * 2. Server responds with an ACK. The xid from this ACK is used to lookup the
 *    pending transaction from the mci_v4_pending_txn table. Once the object is
 *    found, it is removed from the pending table and inserted into the
 *    completed table (mci_v4_completed_txn, keyed by client ID) and the dynamic
 *    IP table (mci_v4_dyn_ip, keyed by IP address).
 *
 * 3. An outgoing packet that goes through the ip-nospoof path will be checked
 *    against the dynamic IP table. Packets that have the assigned DHCP address
 *    as the source IP address will pass the check and be admitted onto the
 *    network.
 *
 * IPv4 notes:
 *
 * If the server never responds with an ACK, there is a timer that is set after
 * the insertion of the transaction into the pending table. When the timer
 * fires, it will check whether the transaction is old (by comparing current
 * time and the txn's timestamp), if so the transaction will be freed. along
 * with this, any transaction in the completed/dyn-ip tables matching the client
 * ID of this stale transaction will also be freed. If the client fails to
 * extend a lease, we want to stop the client from using any IP addresses that
 * were granted previously.
 *
 * A RELEASE message from the client will not cause a transaction to be created.
 * The client ID in the RELEASE message will be used for finding and removing
 * transactions in the completed and dyn-ip tables.
 *
 *
 * For IPv6, the process to acquire a DHCPv6-assigned address is as follows:
 *
 * 1. Client sends REQUEST. The DUID is extracted and stored into a dhcpv6_cid_t
 *    structure. A new transaction structure (dhcpv6_txn_t) is also created and
 *    it will point to the dhcpv6_cid_t. If an existing transaction with a
 *    matching xid is not found, this dhcpv6_txn_t will be inserted into the
 *    mci_v6_pending_txn table (keyed by xid).
 *
 * 2. Server responds with a REPLY. If a pending transaction is found, the
 *    addresses in the reply will be placed into the dhcpv6_cid_t pointed to by
 *    the transaction. The dhcpv6_cid_t will then be moved to the mci_v6_cid
 *    table (keyed by cid). The associated addresses will be added to the
 *    mci_v6_dyn_ip table (while still being pointed to by the dhcpv6_cid_t).
 *
 * 3. IPv6 ip-nospoof will now check mci_v6_dyn_ip for matching packets.
```

```
128  *    Packets with a source address matching one of the DHCPv6-assigned
129  *    addresses will be allowed through.
130  *
131  * IPv6 notes:
132  *
133  * The v6 code shares the same timer as v4 for scrubbing stale transactions.
134  * Just like v4, as part of removing an expired transaction, a RELEASE will be
135  * be triggered on the cid associated with the expired transaction.
136  *
137  * The data structures used for v6 are slightly different because a v6 client
138  * may have multiple addresses associated with it.
139  */

141 /*
142  * These are just arbitrary limits meant for preventing abuse (e.g. a user
143  * flooding the network with bogus transactions). They are not meant to be
144  * user-modifiable so they are not exposed as linkprops.
145  */
146 static ulong_t  dhcp_max_pending_txn = 512;
147 static ulong_t  dhcp_max_completed_txn = 512;
148 static hrtime_t txn_cleanup_interval = 60 * NANOSEC;
 25 static time_t   txn_cleanup_interval = 60;

150 /*
151  * DHCPv4 transaction. It may be added to three different tables
152  * (keyed by different fields).
153  */
154 typedef struct dhcpv4_txn {
155         uint32_t                dt_xid;
156         hrtime_t                dt_timestamp;
 33         time_t                  dt_timestamp;
157         uint8_t                 dt_cid[DHCP_MAX_OPT_SIZE];
158         uint8_t                 dt_cid_len;
159         ipaddr_t                dt_ipaddr;
160         avl_node_t              dt_node;
161         avl_node_t              dt_ipnode;
162         struct dhcpv4_txn       *dt_next;
163 } dhcpv4_txn_t;
______unchanged_portion_omitted_

187 /*
188  * DHCPv6 transaction. Unlike its v4 counterpart, this object gets freed up
189  * as soon as the transaction completes or expires.
190  */
191 typedef struct dhcpv6_txn {
192         uint32_t                dt_xid;
193         hrtime_t                dt_timestamp;
 70         time_t                  dt_timestamp;
194         dhcpv6_cid_t            *dt_cid;
195         avl_node_t              dt_node;
196         struct dhcpv6_txn       *dt_next;
197 } dhcpv6_txn_t;
______unchanged_portion_omitted_

450 /*
451  * Create/destroy a DHCPv4 transaction.
452  */
453 static dhcpv4_txn_t *
454 create_dhcpv4_txn(uint32_t xid, uint8_t *cid, uint8_t cid_len, ipaddr_t ipaddr)
455 {
456         dhcpv4_txn_t    *txn;

458         if ((txn = kmem_zalloc(sizeof (*txn), KM_NOSLEEP)) == NULL)
459                 return (NULL);

461         txn->dt_xid = xid;
```

```
462         txn->dt_timestamp = gethrtime();
339         txn->dt_timestamp = ddi_get_time();
463         if (cid_len > 0)
464                 bcopy(cid, &txn->dt_cid, cid_len);
465         txn->dt_cid_len = cid_len;
466         txn->dt_ipaddr = ipaddr;
467         return (txn);
468 }
______unchanged_portion_omitted_

505 /*
506  * Cleanup stale DHCPv4 transactions.
507  */
508 static void
509 txn_cleanup_v4(mac_client_impl_t *mcip)
510 {
511         dhcpv4_txn_t            *txn, *ctxn, *next, *txn_list = NULL;

513         /*
514          * Find stale pending transactions and place them on a list
515          * to be removed.
516          */
517         for (txn = avl_first(&mcip->mci_v4_pending_txn); txn != NULL;
518             txn = avl_walk(&mcip->mci_v4_pending_txn, txn, AVL_AFTER)) {
519                 if (gethrtime() - txn->dt_timestamp > txn_cleanup_interval) {
396                 if (ddi_get_time() - txn->dt_timestamp >
397                     txn_cleanup_interval) {
520                         DTRACE_PROBE2(found__expired__txn,
521                             mac_client_impl_t *, mcip,
522                             dhcpv4_txn_t *, txn);

524                         txn->dt_next = txn_list;
525                         txn_list = txn;
526                 }
527         }

529         /*
530          * Remove and free stale pending transactions and completed
531          * transactions with the same client IDs as the stale transactions.
532          */
533         for (txn = txn_list; txn != NULL; txn = next) {
534                 avl_remove(&mcip->mci_v4_pending_txn, txn);

536                 ctxn = find_dhcpv4_completed_txn(mcip, txn->dt_cid,
537                     txn->dt_cid_len);
538                 if (ctxn != NULL) {
539                         DTRACE_PROBE2(removing__completed__txn,
540                             mac_client_impl_t *, mcip,
541                             dhcpv4_txn_t *, ctxn);

543                         remove_dhcpv4_completed_txn(mcip, ctxn);
544                         free_dhcpv4_txn(ctxn);
545                 }
546                 next = txn->dt_next;
547                 txn->dt_next = NULL;

549                 DTRACE_PROBE2(freeing__txn, mac_client_impl_t *, mcip,
550                     dhcpv4_txn_t *, txn);
551                 free_dhcpv4_txn(txn);
552         }
553 }

555 /*
556  * Core logic for intercepting outbound DHCPv4 packets.
557  */
558 static boolean_t
```

```
intercept_dhcpv4_outbound(mac_client_impl_t *mcip, ipha_t *ipha, uchar_t *end)
{
        struct dhcp             *dh4;
        uchar_t                 *opt;
        dhcpv4_txn_t            *txn, *ctxn;
        ipaddr_t                ipaddr;
        uint8_t                 opt_len, mtype, cid[DHCP_MAX_OPT_SIZE], cid_len;
        mac_resource_props_t    *mrp = MCIP_RESOURCE_PROPS(mcip);

        if (get_dhcpv4_info(ipha, end, &dh4) != 0)
                return (B_TRUE);

        /* ip_nospoof/allowed-ips and DHCP are mutually exclusive by default */
        if (allowed_ips_set(mrp, IPV4_VERSION))
                return (B_FALSE);

        if (get_dhcpv4_option(dh4, end, CD_DHCP_TYPE, &opt, &opt_len) != 0 ||
            opt_len != 1) {
                DTRACE_PROBE2(mtype__not__found, mac_client_impl_t *, mcip,
                    struct dhcp *, dh4);
                return (B_TRUE);
        }
        mtype = *opt;
        if (mtype != REQUEST && mtype != RELEASE) {
                DTRACE_PROBE3(ignored__mtype, mac_client_impl_t *, mcip,
                    struct dhcp *, dh4, uint8_t, mtype);
                return (B_TRUE);
        }

        /* client ID is optional for IPv4 */
        if (get_dhcpv4_option(dh4, end, CD_CLIENT_ID, &opt, &opt_len) == 0 &&
            opt_len >= 2) {
                bcopy(opt, cid, opt_len);
                cid_len = opt_len;
        } else {
                bzero(cid, DHCP_MAX_OPT_SIZE);
                cid_len = 0;
        }

        mutex_enter(&mcip->mci_protect_lock);
        if (mtype == RELEASE) {
                DTRACE_PROBE2(release, mac_client_impl_t *, mcip,
                    struct dhcp *, dh4);

                /* flush any completed txn with this cid */
                ctxn = find_dhcpv4_completed_txn(mcip, cid, cid_len);
                if (ctxn != NULL) {
                        DTRACE_PROBE2(release__successful, mac_client_impl_t *,
                            mcip, struct dhcp *, dh4);

                        remove_dhcpv4_completed_txn(mcip, ctxn);
                        free_dhcpv4_txn(ctxn);
                }
                goto done;
        }

        /*
         * If a pending txn already exists, we'll update its timestamp so
         * it won't get flushed by the timer. We don't need to create new
         * txns for retransmissions.
         */
        if ((txn = find_dhcpv4_pending_txn(mcip, dh4->xid)) != NULL) {
                DTRACE_PROBE2(update, mac_client_impl_t *, mcip,
                    dhcpv4_txn_t *, txn);
                txn->dt_timestamp = gethrtime();
                txn->dt_timestamp = ddi_get_time();
```

```
                goto done;
        }

        if (get_dhcpv4_option(dh4, end, CD_REQUESTED_IP_ADDR,
            &opt, &opt_len) != 0 || opt_len != sizeof (ipaddr)) {
                DTRACE_PROBE2(ipaddr__not__found, mac_client_impl_t *, mcip,
                    struct dhcp *, dh4);
                goto done;
        }
        bcopy(opt, &ipaddr, sizeof (ipaddr));
        if ((txn = create_dhcpv4_txn(dh4->xid, cid, cid_len, ipaddr)) == NULL)
                goto done;

        if (insert_dhcpv4_pending_txn(mcip, txn) != 0) {
                DTRACE_PROBE2(insert__failed, mac_client_impl_t *, mcip,
                    dhcpv4_txn_t *, txn);
                free_dhcpv4_txn(txn);
                goto done;
        }
        start_txn_cleanup_timer(mcip);

        DTRACE_PROBE2(txn__pending, mac_client_impl_t *, mcip,
            dhcpv4_txn_t *, txn);

done:
        mutex_exit(&mcip->mci_protect_lock);
        return (B_TRUE);
}
______unchanged_portion_omitted_

static dhcpv6_txn_t *
create_dhcpv6_txn(uint32_t xid, dhcpv6_cid_t *cid)
{
        dhcpv6_txn_t    *txn;

        if ((txn = kmem_zalloc(sizeof (dhcpv6_txn_t), KM_NOSLEEP)) == NULL)
                return (NULL);

        txn->dt_xid = xid;
        txn->dt_cid = cid;
        txn->dt_timestamp = gethrtime();
        txn->dt_timestamp = ddi_get_time();
        return (txn);
}
______unchanged_portion_omitted_

/*
 * Cleanup stale DHCPv6 transactions.
 */
static void
txn_cleanup_v6(mac_client_impl_t *mcip)
{
        dhcpv6_txn_t            *txn, *next, *txn_list = NULL;

        /*
         * Find stale pending transactions and place them on a list
         * to be removed.
         */
        for (txn = avl_first(&mcip->mci_v6_pending_txn); txn != NULL;
            txn = avl_walk(&mcip->mci_v6_pending_txn, txn, AVL_AFTER)) {
                if (gethrtime() - txn->dt_timestamp > txn_cleanup_interval) {
                if (ddi_get_time() - txn->dt_timestamp >
                    txn_cleanup_interval) {
                        DTRACE_PROBE2(found__expired__txn,
                            mac_client_impl_t *, mcip,
                            dhcpv6_txn_t *, txn);
                        txn->dt_next = txn_list;
                        txn_list = txn;
                }
        }

        /*
         * Remove and free stale pending transactions.
         * Release any existing cids matching the stale transactions.
         */
        for (txn = txn_list; txn != NULL; txn = next) {
                avl_remove(&mcip->mci_v6_pending_txn, txn);
                release_dhcpv6_cid(mcip, txn->dt_cid);
                next = txn->dt_next;
                txn->dt_next = NULL;

                DTRACE_PROBE2(freeing__txn, mac_client_impl_t *, mcip,
                    dhcpv6_txn_t *, txn);
                free_dhcpv6_txn(txn);
        }

}

/*
 * Core logic for intercepting outbound DHCPv6 packets.
 */
static boolean_t
intercept_dhcpv6_outbound(mac_client_impl_t *mcip, ip6_t *ip6h, uchar_t *end)
{
        dhcpv6_message_t        *dh6;
        dhcpv6_txn_t            *txn;
        dhcpv6_cid_t            *cid = NULL;
        uint32_t                xid;
        uint8_t                 mtype;
        mac_resource_props_t *mrp = MCIP_RESOURCE_PROPS(mcip);

        if (get_dhcpv6_info(ip6h, end, &dh6) != 0)
                return (B_TRUE);

        /* ip_nospoof/allowed-ips and DHCP are mutually exclusive by default */
        if (allowed_ips_set(mrp, IPV6_VERSION))
                return (B_FALSE);

        mtype = dh6->d6m_msg_type;
        if (mtype != DHCPV6_MSG_REQUEST && mtype != DHCPV6_MSG_RENEW &&
            mtype != DHCPV6_MSG_REBIND && mtype != DHCPV6_MSG_RELEASE)
                return (B_TRUE);

        if ((cid = create_dhcpv6_cid(dh6, end)) == NULL)
                return (B_TRUE);

        mutex_enter(&mcip->mci_protect_lock);
        if (mtype == DHCPV6_MSG_RELEASE) {
                release_dhcpv6_cid(mcip, cid);
                goto done;
        }
        xid = DHCPV6_GET_TRANSID(dh6);
        if ((txn = find_dhcpv6_pending_txn(mcip, xid)) != NULL) {
                DTRACE_PROBE2(update, mac_client_impl_t *, mcip,
                    dhcpv6_txn_t *, txn);
                txn->dt_timestamp = gethrtime();
                txn->dt_timestamp = ddi_get_time();
                goto done;
        }
        if ((txn = create_dhcpv6_txn(xid, cid)) == NULL)
                goto done;
```

```
        cid = NULL;
        if (insert_dhcpv6_pending_txn(mcip, txn) != 0) {
                DTRACE_PROBE2(insert__failed, mac_client_impl_t *, mcip,
                    dhcpv6_txn_t *, txn);
                free_dhcpv6_txn(txn);
                goto done;
        }
        start_txn_cleanup_timer(mcip);

        DTRACE_PROBE2(txn__pending, mac_client_impl_t *, mcip,
            dhcpv6_txn_t *, txn);

done:
        if (cid != NULL)
                free_dhcpv6_cid(cid);

        mutex_exit(&mcip->mci_protect_lock);
        return (B_TRUE);
}
```

______***unchanged_portion_omitted_***

```
/*
 * Timer for cleaning up stale transactions.
 */
static void
txn_cleanup_timer(void *arg)
{
        mac_client_impl_t       *mcip = arg;

        mutex_enter(&mcip->mci_protect_lock);
        if (mcip->mci_txn_cleanup_tid == 0) {
                /* do nothing if timer got cancelled */
                mutex_exit(&mcip->mci_protect_lock);
                return;
        }
        mcip->mci_txn_cleanup_tid = 0;

        txn_cleanup_v4(mcip);
        txn_cleanup_v6(mcip);

        /*
         * Restart timer if pending transactions still exist.
         */
        if (!avl_is_empty(&mcip->mci_v4_pending_txn) ||
            !avl_is_empty(&mcip->mci_v6_pending_txn)) {
                DTRACE_PROBE1(restarting__timer, mac_client_impl_t *, mcip);

                mcip->mci_txn_cleanup_tid = timeout(txn_cleanup_timer, mcip,
                    drv_usectohz(txn_cleanup_interval / (NANOSEC / MICROSEC)));
                    drv_usectohz(txn_cleanup_interval * 1000000));
        }
        mutex_exit(&mcip->mci_protect_lock);
}

static void
start_txn_cleanup_timer(mac_client_impl_t *mcip)
{
        ASSERT(MUTEX_HELD(&mcip->mci_protect_lock));
        if (mcip->mci_txn_cleanup_tid == 0) {
                mcip->mci_txn_cleanup_tid = timeout(txn_cleanup_timer, mcip,
                    drv_usectohz(txn_cleanup_interval / (NANOSEC / MICROSEC)));
                    drv_usectohz(txn_cleanup_interval * 1000000));
        }
}
```

______***unchanged_portion_omitted_***