Appendix C DLP PARTITIONING EXAMPLE

DNS protocol is usually considered as a commonly used protocol to interpret the domain names on the Internet thus is rarely prohibited by firewalls. In a DNS Tunneling attack, the attacker tries to bypass the firewall by encapsulating the malicious content in DNS traffic. The DNS Tunneling DLP tries to detect DNS Tunneling attacks. We use this DLP as a use case because the DNS Tunneling detection techniques have been well studied in lots of literature [27, 32, 40, 41], and there is also a DLP implemented by Bro scripting language [7]. We have implemented a C version of DNS Tunneling DLP based on existing Bro DNS Tunnels detection scripts. The DLP involves both header-based DL and payload-based DL. Thus, it would be a good use case to demonstrate our approach for partitioning the DLP.

The code shown in Code 1 is an implementation of the DNS Tunneling DLP in C language. The Tunneling_Detection is a C interface stub that will be called back by the NE when a DNS packet is received. The parameters sip and length are header fields of incoming packets, while the query is the payload information indicating the domain name being queried by the DNS packet. The basic idea of the DLP is as follows: The hosts is a pointer of a linked list. Each element in the list is a structure recording which IP address this element is associated with and the number of DNS packets being counted by the DLP. Each time a DNS packet is received, the DLP increments the count accordingly (lines 26 and 27). The DLP first checks whether a host has generated too many DNS packets in a period of time (line 28). If this is true, this host is reported as an attacker (lines 29-31). Otherwise, the DLP checks the length of the DNS packet (line 33). If the packet's size exceeds a threshold, the DLP further checks the payload of the packet (lines 34-41); otherwise, this is a normal packet.

Step1 & Step2: We use Frama-C to compute the forward slice of query from the beginning of the DLP (line 25). Note that Frama-C has already computed the PDG internally for its program slicing. The resulting statements of the payload-based DLP slice are lines 35, 36, 39, 40, 41, and 45.

Step3: To construct the header-based DLP, we replace lines 35, 36, 39, 40, 41, and 45 with forward statements. Then, we identify the variables that should be tracked. According to Algorithm 1, variable length and i have been used by both header-based and payload-based DLPs. Therefore, those two variables should be tracked. We find that length has never been updated (i.e., never appear on the left-hand-side). As a result, only i is encapsulated into the packet and sent to the payload-based DLP.

Step4: To construct the payload-based DLP, we first add the initialization statements for the variables storing intermediate results. In this case, we only add a statement to assign the value to variable i. Then, right after the initialization statements, we need the switch-case statement to retrieve the breakpoints. In this case, the switch-case statement checks the label and maps each label to lines 35, 36, 39, 40, 41, and 45, respectively.

Step5: Once we have PDGs of both header-based and payload-based DLPs, we can use tools to reverse the PDGs into the source code. In our case, instead of generating PDGs, Frama-C computes the PDG internally and utilizes the PDG to compute the forward program slice. Finally, Frama-C emits the source code of the resulting slice directly. We write a parser using Flex [23] and Bison [13] to do the construction in *Step3* and *Step4*. In practice, most modern compilers can make optimization of the source code by removing unused variables and unreachable statements.

```
1  int T1 = 100; /*frequency threshold*/
2  int T2 = 53; /*length threshold*/
3  int T3 = 0.3; /*numeric threshold*/
4  struct host {
5     unit ip;
6     int count;
7     struct host* next;
8  };
9  struct host* hosts;
10 struct host* find_host(uint ip) {
11    struct host* h = hosts;
12    while (h) {
13       if (h->ip == ip) return h;
14       h = h->next;
15    }
16    h = malloc(sizeof(struct host));
17    h->ip = ip;
18    h->count = 0;
19    h->next = hosts->next;
20    hosts->next = h;
21    return h;
22 }
23
24 int Tunneling_Detection(uint sip,int length,char* query){
25    int num_count = 0, i = 0;
26    struct host* h = find_host(sip);
27    h->count += 1;
28    if (h->count > T1) {
29       h->count = 0;
30       printf("DNS Tunnel Detected!\n");
31       return 1;
32    } else {
33       if (length > T2) {
34          for (i = 0; i < length; i++) {
35             if (query[i]>='0' && query[i]<='9'){
36                num_count += 1;
37             }
38          }
39          if (num_count > length * T3) {
40             printf("DNS Tunnel Detected!\n");
41             return 2;
42          }
43       }
44    }
45    return 0;
46 }
```

Code 1: Simplified DNS Tunneling Detection Program