Conversion from Attack Graphs to Attack Trees

Example attack graph:
Let us consider the following artificial system composed of **host1**, **host2** and **host3** for illustration purposes. Outside access is enabled on host1 and host2. In addition, host1 and host2 are connected to host3. The privilege levels considered in this example are only **NONE(lowest)** and **ADMIN(highest)**. However, this example can be easily extended to accommodate also VOS(USER), VOS(ADMIN) and USER privilege lebels. After a vulnerability scanner is run on the attack graph, the following vulnerabilities are detected. **CVE1(NONE → ADMIN)** and **CVE2(NONE → ADMIN)** on host1, **CVE3 (NONE → ADMIN)** on host2 and **CVE4 (NONE → ADMIN)** on host3. The attacker is always modeled as "outside" and it always has an ADMIN privilege. The resulting attack graph after the attack graph generation procedure is displayed on Figure 1.

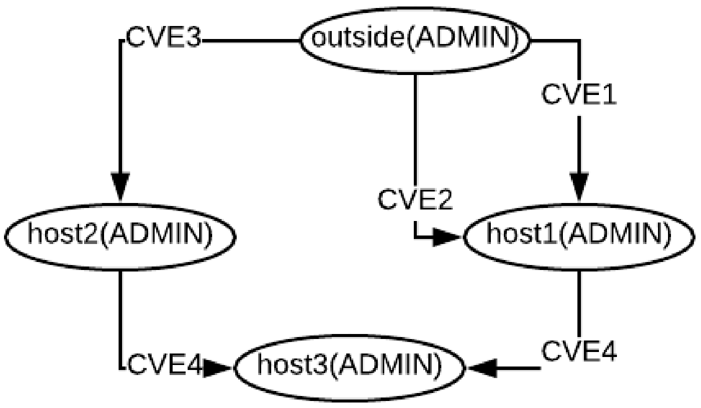

Figure 1. Example attack graph

Resulting attack tree:
Let us say that we want to convert the above attack graph to an attack tree. In this process we have to select a goal host. In this example that would be host3. The resulting attack tree is on Figure 2. The attack graph contains 5 kind of nodes: **CVE nodes, host nodes, cond nodes, adj_cond nodes and vul_cond nodes**. CVE nodes describe the vulnerabilities that could be exploited in a given system and they are always connected to vul_cond nodes in an OR relation. Host nodes describe the current neighbouring host with its required privilege needed to execute an attack. The rest three nodes are intermediate nodes to model additional conditions. Adj_cond nodes represents the adjacency condition i.e. in order for a vulnerability in a given node to be exploited, the attacker has to have an access to a neighbouring node. Vul_cond nodes represent the vulnerability condition, i.e., at least one vulnerability has to be present in a goal container in order for it to be exploited. Cond nodes are a combination of adj_cond and vul_cond nodes. They model different ways that a container can be attacked. The number of nodes in the attack tree is calculated as follows:

n_nodes_attack_tree = n_edges_graph + n_unique_edges_graph[*] ** 4 + 1(goal container)*

[*]*n_unique_edges_graph – In the case where we have parallel edges between two nodes, we only count one occurrence.*

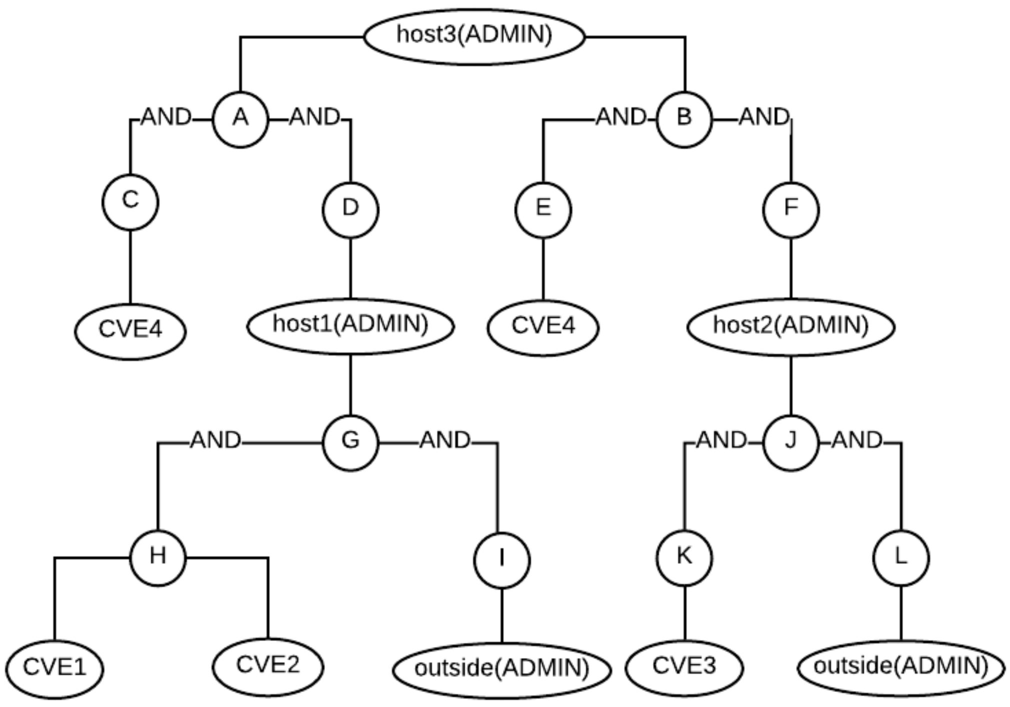

Figure 2. Resulting attack tree

The resulting attack tree is displayed in Figure 2. For display and clarity purposes, some of the node names are referenced bellow.

The attack tree is composed of 5 kind of nodes:

- CVE nodes (always a leaf)
  - CVE1, CVE2, CVE3, CVE4
- host(privilege)
  - outside(ADMIN), host1(ADMIN), host2(ADMIN), host3(ADMIN)
- 3 condition nodes:
  - cond → always a combination between adj_cond and vul_cond in AND relation
    - A – cond_host1(ADMIN)_host3(ADMIN)
    - B – cond_host2(ADMIN)_host3(ADMIN)
    - G – cond_outside(ADMIN)_host1(ADMIN)
    - J – cond_outside(ADMIN)_host2(ADMIN)
  - vul_cond →vulnerability condition is fulfilled in OR relation
    - C – vul_cond_host1(ADMIN)_host3(ADMIN)
    - E – vul_cond_host2(ADMIN)_host3(ADMIN)
    - H – vul_cond_outside(ADMIN)_host1(ADMIN)
    - K – vul_cond_outside(ADMIN)_host2(ADMIN)
  - adj_cond → adjacency condition is fulfilled, neighbouring node from where an attack is initialized
    - D – vul_cond_host1(ADMIN)_host3(ADMIN)
    - F – vul_cond_host2(ADMIN)_host3(ADMIN)
    - I – vul_cond_outside(ADMIN)_host1(ADMIN)
    - L – vul_cond_outside(ADMIN)_host2(ADMIN)

Procedure:
1) We start from the node defined as a goal host for the attack graph generation
2) We look at its neighbours. We only take the nodes that have to goal host as a termination host.
3) For each of the selected neighbours (neighbouring nodes in the attack graph) we add a cond node.
4) For each cond node we add an vul_cond and adj_cond nodes.
5) We add the vulnerabilities of a neighouring node (edges in the attack graph) to the vul_cond node in an OR relation.
6) We add the neighbouring node to the adj_cond node with an edge.
7) We repeat this procedure for all of the unvisited nodes until we reach to outside(ADMIN) node

Substructures:
In order two ensure that the attack tree generation conforms to the options in the resulting attack graph, we had to consider elementary substructures that occur in the produced attack graph:
1) Trivial case. A node can be attacked from a different node by exploiting a single vulnerability (Figure 3).

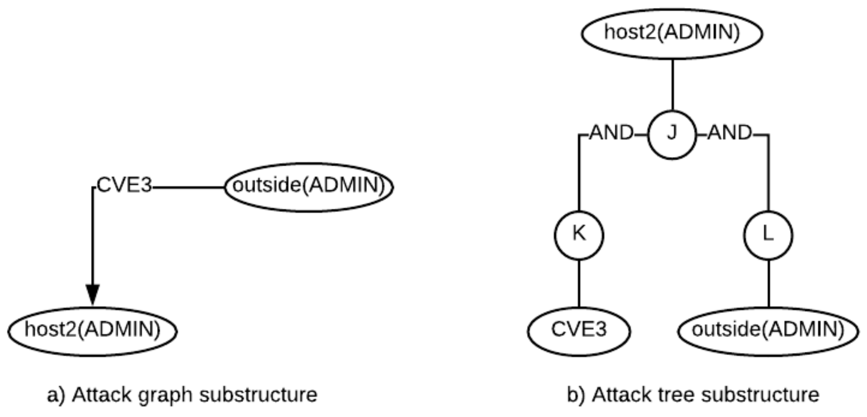

Figure 3. Example attack graph and attack tree substructures

2) A host can be attacked from two different hosts (Figure 4). We can notice an extension in the structure with an additional cond node in comparison to case 1)

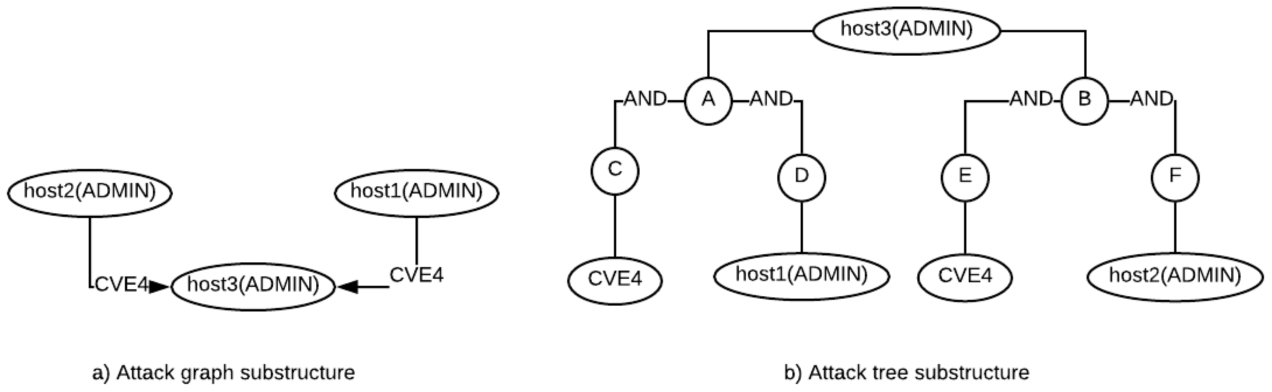

Figure 4. Example attack graph and attack tree substructures

3) A host can be attacked by exploiting multiple different vulnerabilities (Figure 5). This also models a subcase where multiple vulnerabilities are exploited from single previously conquered node.

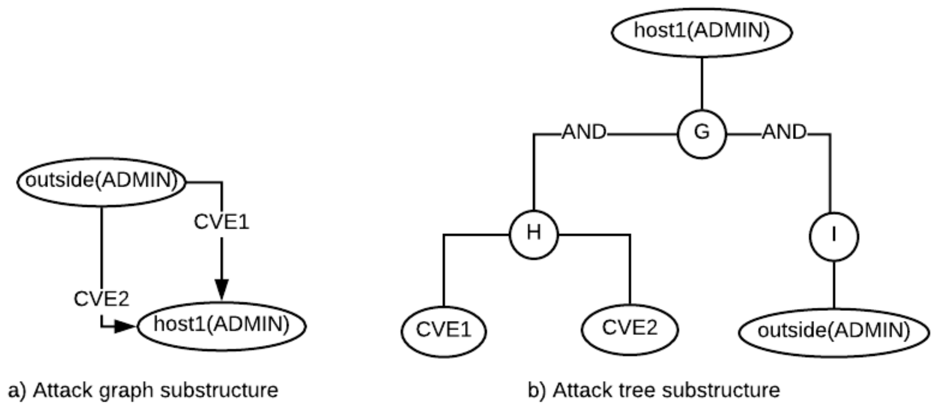

Figure 5. Example attack graph and attack tree substructures

4) By chaining the substructures together, we get the complete attack tree (Figure 2).

*Reducing redundancy - We noticed some possible redundancy in the attack tree description. The adj_cond nodes can be removed and we can add the host nodes directly to the cond nodes.