Third Challenge Write-Up - Security II 2021/2022

Hernest Serani - 877028

April 12, 2022

Introduction

This simple script can be used to verify the reachability of simple *ARBAC* rules. It is composed of a parser which parses an *.arbac* file written into a specific format (as explained in the description pdf), and saves all it's elements into adeguate data structures. Once the parsing is done, the analyzer starts analyzing the reachability of the target role.

First it executes the forward and backward slicing on the initial parsed data, in order to reduce the state space. The importance will be discussed in the Oberservations chapter of this report.

It is developed for the purpose of solving the third challenge of the Security 2 course. In roughly 5 minutes it finds the reachabilities for all the 8 policies of the challenge, and so, obtain the flag as the concatenation of the reachabilities expressed as zero or one.

Implementation

The pseudocode demonstrates the algorithm which performs the role reachability check. The idea is pretty straight forward, we start with an initial to-visit states which consists of the current user-role assignments (the one parsed from the policy file). The initial already-visited states is empty.

Algorithm 1 Verifying Reachability

```
$found \leftarrow False$
$TV \leftarrow [UA]$                                              ▷ To-Visit States
$AV \leftarrow \emptyset$                                         ▷ Already-Visited States
*Apply forward slicing*
*Apply backward slicing*
while $|TV| > 0$ and $!found$ do
    $cs \leftarrow TV.pop()$
    if $hash(cs) \in AV$ then                                     ▷ State already visited
        *continue*
    end if
    if $goal \in UR(cs)$ then                                     ▷ Goal role is reachable
        $found \leftarrow True$
    end if
    $AV \leftarrow AV \cup \{cs\}$
    for $ca \in CA$ do
        for $user \in UR(cs)$ do
            for $target \in users$ do
                assign(target, ca, cs)                            ▷ Try to assign the role
            end for
        end for
    end for
    for $cr \in CR$ do
        for $user \in UR(cs)$ do
            for $target \in users$ do
                revoke(target, cr, cs)                            ▷ Try to revoke the role
            end for
        end for
    end for
end while
```

Once we apply forward and backward slicing (in order to try to reduce the states), we loop through all the to-visit states, unless we find that the goal role is reachable. Then we try in a bruteforce approach to assign and revoke the rules, to all the users for each current user-role assignment. For each potential target, we try to assign the roles which are mentioned on the Can-Assign parsed data structure. The user which assigns this role to the target, is the one mentioned on the first part of the rule. The same approach is applied for the revocation part.

The other important functions are the one which assigns a role to a target, and the one which revokes it. To implement these rules, I followed the algorithm given during the lecture.

The assignment of a new role is expressed as follows:

$$\frac{\begin{array}{c}(u_a, r_a) \in UR \quad (r_a, R_p, R_n, r_t) \in CA \\ R_p \subseteq UR(u_t) \quad R_n \cap UR(u_t) = \emptyset \quad r_t \notin UR(u_t)\end{array}}{UR \rightarrow_{\mathcal{P}} UR \cup \{(u_t, r_t)\}}$$

The revocation of a new role is expressed as follows:

$$\frac{(u_a, r_a) \in UR \quad (r_a, r_t) \in CR \quad r_t \in UR(u_t)}{UR \rightarrow_{\mathcal{P}} UR \setminus \{(u_t, r_t)\}}$$

Python is pretty good at expressing mathematical notations, so this is the reason I chose to develop it in Python.

For the forward and backward slicing, I again followed the algorithms explained during the lecture.

Forward slicing:

$$S_0 = \{r \in R \mid \exists u \in U : (u, r) \in UR\}$$
$$S_i = S_{i-1} \cup \{r_t \in R \mid (r_a, R_p, R_n, r_t) \in CA \wedge R_p \cup \{r_a\} \subseteq S_{i-1}\}$$

Backward slicing:

$$S_0 = \{r_g\}$$
$$S_i = S_{i-1} \cup \{R_p \cup R_n \cup \{r_a\} \mid (r_a, R_p, R_n, r_t) \in CA \wedge r_t \in S_{i-1}\}$$

Oberservations

An important aspect is the complexity of this algorithm. It is actually PSPACE-complete. For this reason we may really consider to implement/apply some auxilliary algorithms, in order to make everything efficient. One of these which is implemented in this solution is the slicing algorithm (both backward and forward slicing).

During the output of the program, we can see how many possible states it reduces. Furthermore, that number is not the real states which the algorithm should explore, rather it is just an upperbound limit. Internally as explained above, the algorithm takes note of the already visited states, in order to not revisit them, which may also cause a potential infinite loop. For this reason, we don't know the exact complexity, but we can define an upper bound of the states as $O(2^{|R||U|})$.

The practical benefit of the slicing can be seen during the output, i.e, it reduced the states of a policy from 1427247692705959881058285969449495136382746624 to 1180591620717411303424, which can be considered as a huge advantage. The reductions that are performed by the slicing algorithms are on the number of Can-Assign and Can-Revoke rules, and on the number of roles.