An Algorithm to Find Optimal Attack Paths in Nondeterministic Scenarios

Carlos Sarraute
Core Security Technologies
and ITBA
Buenos Aires, Argentina
carlos@corest.com

Gerardo Richarte
Core Security Technologies
Buenos Aires, Argentina
gera@corest.com

Jorge Lucángeli Obes
Universidad de Buenos Aires
Argentina
jlucangeli@dc.uba.ar

ABSTRACT

As penetration testing frameworks have evolved and have become more complex, the problem of controlling automatically the pentesting tool has become an important question. This can be naturally addressed as an *attack planning* problem. Previous approaches to this problem were based on modeling the actions and assets in the PDDL language, and using off-the-shelf AI tools to generate attack plans. These approaches however are limited. In particular, the planning is classical (the actions are deterministic) and thus not able to handle the uncertainty involved in this form of attack planning. We herein contribute a planning model that does capture the uncertainty about the results of the actions, which is modeled as a probability of success of each action. We present efficient planning algorithms, specifically designed for this problem, that achieve industrial-scale runtime performance (able to solve scenarios with several hundred hosts and exploits). These algorithms take into account the probability of success of the actions and their expected cost (for example in terms of execution time, or network traffic generated). We thus show that probabilistic attack planning can be solved efficiently for the scenarios that arise when assessing the security of large networks. Two "primitives" are presented, which are used as building blocks in a framework separating the overall problem into two levels of abstraction. We also present the experimental results obtained with our implementation, and conclude with some ideas for further work.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Plan execution, formation, and generation*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*; K.6.m [**Management of Computing and Information Systems**]: Miscellaneous—*Security*

AISec'11, October 21, 2011, Chicago, Illinois, USA.

General Terms

Security

Keywords

Network security, exploit, automated pentesting, attack planning

1. INTRODUCTION

Penetration testing has become one of the most trusted ways of assessing the security of networks large and small. The result of a penetration test is a repeatable set of steps that result in the compromise of particular assets in the network. Penetration testing frameworks have been developed to facilitate the work of penetration testers and make the assessment of network security more accessible to non-expert users [6]. The main tools available are the commercial products Core Impact (since 2001), Immunity Canvas (since 2002), and the open source project Metasploit (launched in 2003, owned by Rapid7 since 2009). These tools have the ability to launch actual exploits for vulnerabilities, contributing to expose risk by conducting an attack in the same way an external attacker would [3].

As pentesting tools have evolved and have become more complex – covering new attack vectors, and shipping increasing numbers of exploits and information gathering techniques – the problem of controlling the pentest framework successfully has become an important question. A computer-generated plan for an attack would isolate the user from the complexity of selecting suitable exploits for the hosts in the target network. In addition, a suitable model to represent these attacks would help to systematize the knowledge gained during manual penetration tests performed by expert users, making pentesting frameworks more accessible to non-experts.

A natural way to address this issue is as an *attack planning* problem. This problem was introduced to the AI planning community by Boddy *et al.* as the "Cyber Security" domain [5]. In the pentesting industry, Lucangeli *et al.* proposed a solution based on modeling the actions and assets in the PDDL language,[1] and using off-the-shelf planners to generate attack plans [16]. Herein we are concerned with the specific context of regular automated pentesting, as in "Core Insight Enterprise" tool. We will use the term "attack planning" in that sense.

[1]PDDL stands for Planning Domain Definition Language. Refer to [10] for a specification of PDDL 2.1.

Recently, a model based on partially observable Markov decision processes (POMDP) was proposed, in part by one of the authors [25]. This grounded the attack planning problem in a well-researched formalism, and provided a precise representation of the attacker's uncertainty with respect to the target network. In particular, the information gathering phase was modeled as an integral part of the planning problem. However, as the authors show, this solution does not scale to medium or large real-life networks.

In this paper, we take a different direction: the uncertainty about the results of the actions is modeled as a *probability of success* of each action, whereas in [25] the uncertainty is modeled as a distribution of probabilities over the states. This allows us to produce an efficient planning algorithm, specifically designed for this problem, that achieves industrial-scale runtime performance.

Of course planning in the probabilistic setting is far more difficult than in the deterministic one. We do not propose a general algorithm, but a solution suited for the scenarios that need to be solved in a real world penetration test. The computational complexity of our planning solution is $\mathcal{O}(n \log n)$, where n is the total number of actions in the case of an attack tree (with fixed source and target hosts), and $\mathcal{O}(M^2 \cdot n \log n)$ where M is the number of machines in the case of a network scenario. With our implementation, we were able to solve planning in scenarios with up to 1000 hosts distributed in different networks.

We start with a brief review of the attack model in Section 2, then continue with a presentation of two "primitives" in Sections 3 and 4. These primitives are applied in more general settings in Sections 5 and 6. Section 7 shows experimental results from the implementation of these algorithms. We conclude with some ideas for future work.

2. THE ATTACK MODEL

We provide below some background on the conceptual model of computer attacks that we use, for more details refer to [4, 11, 22, 24]. This model is based on the concepts of assets, goals, agents and actions. In this description, an attack involves a set of agents, executing sequences of actions, obtaining assets (which can be information or actual modifications of the real network and systems) in order to reach a set of goals.

An *asset* can represent anything that an attacker may need to obtain during the course of an attack, including the actual goal. Examples of assets: information about the Operating System (OS) of a host H; TCP connectivity with host H on port P; an Agent installed on a given host H. To *install an agent* means to break into a host, take control of its resources, and eventually use it as pivoting stone to continue the attack by launching new actions based from that host.

The *actions* are the basic steps which form an attack. Actions have requirements (also called preconditions) and a result: the asset that will be obtained if the action is successful. For example, consider the exploit *IBM Tivoli Storage Manager Client Remote Buffer Overflow*[2] for the vulnerabilities in *dsmagent* described by CVE-2008-4828. The result of this action is to install an agent, and it requires that the OS of the target host is Windows 2000, Windows XP, Solaris 10, Windows 2003, or AIX 5.3. In this model, all the exploits (local, remote, client-side, webapps) are represented as actions. Other examples of actions are: TCP Network Discovery, UDP Port Scan, DCERPC OS Detection, TCP Connectivity Probe.

The major differences between the attack model used in this work and the *attack graphs* used in [2, 14, 15, 20, 23, 27] are twofold: to improve the realism of the model, we consider that the actions can produce numerical effects (for example, the expected running time of each action); and that the actions have a probability of success (which models the uncertainty about the results of the action).

[2]The particular implementations that we have studied are the exploit modules for Core Impact and Core Insight Enterprise, although the same model can be applied to other implementations, such as Metasploit.

Deterministic Actions with Numerical Effects

In the deterministic case, the actions and assets that compose a specific planning problem can be successfully represented in the PDDL language. This idea was proposed in [26] and further analyzed in [16]. The assets are represented as PDDL predicates, and the actions are translated as PDDL operators. The authors show how this PDDL representation allowed them to integrate a penetration testing tool with an external planner, and to generate attack plans in realistic scenarios. The planners used – Metric-FF [13] and SGPlan [7] – are state-of-the-art planners able to handle numerical effects.

Fig. 1 shows an example of a PDDL action: an exploit for the IBM Tivoli vulnerability, that will attempt to install an agent on target host t from an agent previously installed on the source host s. To be successful, this exploit requires that the target runs a specific OS, has the service `mil-2045-47001` running and listening on port 1581.

```
(:action IBM_Tivoli_Storage_Manager_Client_Exploit
:parameters (?s - host ?t - host)
:precondition (and
  (compromised ?s)
  (and (has_OS ?t Windows)
    (has_OS_edition ?t Professional)
    (has_OS_servicepack ?t Sp2)
    (has_OS_version ?t WinXp)
    (has_architecture ?t I386))
  (has_service ?t mil-2045-47001)
  (TCP_connectivity ?s ?t port1581)
)
:effect(and
  (installed_agent ?t high_privileges)
  (increase (time) 4)
))
```

Figure 1: Exploit represented as PDDL action.

The average running times of the exploits are measured by executing all the exploits of the penetration testing tool in a testing lab. More specifically, in Core's testing lab there are more than 748 virtual machines with different OS and installed applications, where all the exploits of Core Impact are executed every night [21].

Actions' Costs

The execution of an action has a multi-dimensional cost. We detail below some values that can be measured (and optimized in an attack):

Execution time: Average running time of the action.

Network traffic: The amount of traffic sent over the network increases the level of noise produced.

IDS detection: Logs lines generated and alerts triggered by the execution of the action increase the noise produced.

Host resources: The execution of actions will consume resources of both the local and remote host, in terms of CPU, RAM, hard disc usage, etc.

Traceability of the attack: Depends on the number of intermediate hops and topological factors.

Zero-day exploits: Exploits for vulnerabilities that are not publicly known are a valuable resource, that should be used only when other exploits have failed (the attacker usually wants to minimize the use of "0-days").

In our experiments, we have chosen to optimize the expected execution time. In the context of regular penetration tests, minimizing the expectation of total execution time is a way of maximizing the amount of exploits successfully launched in a fixed time frame (pentests are normally executed in a bounded time period).

However, the same techniques can be applied to any other scalar cost, for example to minimize the noise produced by the actions (and the probability of being detected).

Probabilistic Actions

Another way to add realism to the attack model is to consider that the actions are nondeterministic. This can be modeled by associating probabilities to the outcomes of the actions. In the case of an exploit, the execution of the exploit can be successful (in that case the attacker takes control of the target machine) or a failure. This is represented by associating a *probability of success* to each exploit.

The probability of success is conditional: it depends on the environment conditions. For example, the IBM Tivoli exploit for CVE-2008-4828 is more reliable (has a higher probability of success) if the OS is Solaris since it has no heap protection, the stack is not randomized and is executable by default. Alternatively, the exploit is less reliable (has a lower probability of success) if the OS is Windows XP SP2 or Windows 2003 SP1, with Data Execution Prevention (DEP) enabled. On Windows Vista, the addition of Address Space Layout Randomization (ASLR) makes the development of an exploit even more difficult, and diminishes its probability of success. In practice, the probability of success of each exploit is measured by exhaustively executing the exploit against a series of targets, covering a wide range of OS and application versions.

Although it improves the realism of the model, considering probabilistic actions also makes the planning problem more difficult. Using general purpose probabilistic planners did not work as in the deterministic case; for instance, we experimented with Probabilistic-FF [8] with poor results, since it was able to find plans in only very small cases.

In the rest of this paper, we will study algorithms to find optimal attack paths in scenarios of increasing difficulty. We first describe two primitives, and then apply them in the context of regular automated pentesting. In these scenarios we make an additional hypothesis: the independence of the actions. Relaxing this hypothesis is a subject for future work.

3. THE *CHOOSE* PRIMITIVE

We begin with the following basic problem. Suppose that the attacker (i.e. pentester) wants to gain access to the credit cards stored in a database server H by installing a system agent. The attacker has a set of n remote exploits that he can launch against that server. These exploits result in the installation of a system agent when successful (see Fig. 2).

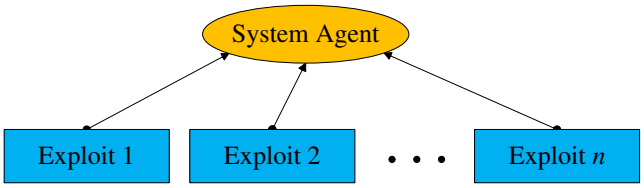

Figure 2: Multiple exploits may install a System agent (on the target host).

In this scenario, the attacker has already performed information gathering about the server H, collecting a list of open/closed ports, and running an OS detection action such as Nmap. The pentesting tool used provides statistics on the probability of success and expected running time for each exploit in the given conditions.[3] The attacker wants to minimize the expected execution time of the whole attack. A more general formulation follows:

Problem 1. Let $\mathfrak{g}$ be a fixed goal, and let $\{A_1, \ldots, A_n\}$ be a set of n independent actions whose result is $\mathfrak{g}$. Each action A_k has a probability of success p_k and expected cost t_k. Actions are executed until an action is successful and provides the goal $\mathfrak{g}$ (or all the actions fail).
Task: Find the order in which the actions must be executed in order to minimize the expected total cost.

We make the simplifying assumption that the probability of success of each action is independent from the others. If the actions are executed in the order $A_1, \ldots, A_n$, using the notation $\overline{p_i} = 1 - p_i$, the expected cost can be written as

$$T_{\{1\ldots n\}} = t_1 + \overline{p_1}\, t_2 + \ldots + \overline{p_1}\,\overline{p_2} \ldots \overline{p_{n-1}}\, t_n. \qquad (1)$$

The probability of success is given by

$$P_{\{1\ldots n\}} = p_1 + \overline{p_1}\, p_2 + \overline{p_1}\,\overline{p_2}\, p_3 + \ldots + \overline{p_1} \ldots \overline{p_{n-1}}\, p_n,$$

and the complement $\overline{P_{\{1\ldots n\}}} = \overline{p_1}\,\overline{p_2} \ldots \overline{p_n}$. In particular this shows that the total probability of success does not depend on the order of execution.

Even though this problem is very basic, we didn't find references to its solution. This is why we give below some details on the solution that we found.

LEMMA 1. *Let $A_1, \ldots, A_n$ be actions such that $t_1/p_1 \leqslant t_2/p_2 \leqslant \ldots \leqslant t_n/p_n$. Then*

$$\frac{T_{\{1\ldots n-1\}}}{P_{\{1\ldots n-1\}}} \leqslant \frac{t_n}{p_n}.$$

PROOF. We prove it by induction. The case with two actions is trivial, since we know by hypothesis that $t_1/p_1 \leqslant t_2/p_2$. For the inductive step, suppose that the proposition

[3]In our experiments we used the database of tests of Core Impact and Core Insight Enterprise.

holds for $n-1$ actions. Consider the first three actions A_1, A_2, A_3. The inequality

$$\frac{T_{\{12\}}}{P_{\{12\}}} \leqslant \frac{t_3}{p_3}$$

holds if and only if $t_2/p_2 \leqslant t_3/p_3$. So the first two actions can be considered as a single action A_{12} with expected cost (e.g. running time) $T_{\{12\}}$ and probability of success $P_{\{12\}}$. We have reduced to the case of $n-1$ actions, and we can use the induction hypothesis to conclude the proof. □

PROPOSITION 1. *A solution to Problem 1 is to sort the actions according to the coefficient t_k/p_k (in increasing order), and to execute them in that order. The complexity of finding an optimal plan is thus $\mathcal{O}(n \log n)$.*

PROOF. We prove it by induction. We begin with the case of two actions A_i and A_j such that $t_i/p_i \leqslant t_j/p_j$. It follows easily that $-p_i t_j \leqslant -p_j t_i$ and that

$$t_i + (1-p_i)\, t_j \leqslant t_j + (1-p_j)\, t_i.$$

For the inductive step, suppose for the moment that the actions are numbered so that $t_1/p_1 \leqslant \ldots \leqslant t_n/p_n$, and that the proposition holds for all sets of $n-1$ actions. We have to prove that executing A_1 first is better that executing any other action A_k for all $k \neq 1$. We want to show that

$$\begin{aligned} & t_1 + \sum_{2 \leqslant i \leqslant n} t_i \cdot \prod_{1 \leqslant j \leqslant i-1} \overline{p_j} \\ \leqslant\ & t_k + \sum_{1 \leqslant i \leqslant n,\, i \neq k} t_i \cdot \overline{p_k} \cdot \prod_{1 \leqslant j \leqslant i-1,\, j \neq k} \overline{p_j}. \end{aligned}$$

Notice that in the two previous sums, the coefficients of $t_{k+1}, \ldots, t_n$ are equal in both expressions. They can be simplified, and using notations previously introduced, the inequality can be rewritten

$$T_{\{1\ldots k-1\}} + \overline{P_{\{1\ldots k-1\}}}\, t_k \leqslant t_k + \overline{p_k}\, T_{\{1\ldots k-1\}}$$

which holds if and only if

$$\frac{T_{\{1\ldots k-1\}}}{P_{\{1\ldots k-1\}}} \leqslant \frac{t_k}{p_k}$$

which is true by Lemma 1. We have reduced the problem to sorting the coefficients t_k/p_k. The complexity is that of making the n divisions t_k/p_k and sorting the coefficients. Thus it is $\mathcal{O}(n + n \log n) = \mathcal{O}(n \log n)$. □

We call this the *choose* primitive because it tells you, given a set of actions, which action to choose first: the one that has the smallest t/p value. In particular, it says that you should execute first the actions with smaller cost (e.g. runtime) or higher probability of success, and precisely which is the trade-off between these two dimensions.

The problem of choosing the order of execution within a set of exploits is very common in practice. In spite of that, the automation methods currently implemented in penetration testing frameworks offer an incomplete solution,[4] over which the one proposed here constitutes an improvement.

[4]As of July 2011, Immunity Canvas [1] doesn't provide automated execution of exploits; Metasploit [17] has a feature called *"autopwn"* that launches all the exploits available for the target ports in arbitrary order; Core Impact Pro launches first a set of *"fast"* exploits and then *"brute-force"* exploits [26], but arbitrary order is used within each set; Core Insight Enterprise uses planning techniques based on a PDDL description [16] that takes into account the execution time but not the probability of success of the exploits.

4. THE *COMBINE* PRIMITIVE

Predefined Strategies

We now consider the slightly more general problem where the goal $\mathfrak{g}$ can be obtained by predefined strategies. We call *strategy* a group of actions that must be executed in a specific order. The strategies are a way to incorporate the expert knowledge of the attacker in the planning system (cf. the opening moves in chess). This idea has been used in the automation of pentesting tools, see [26].

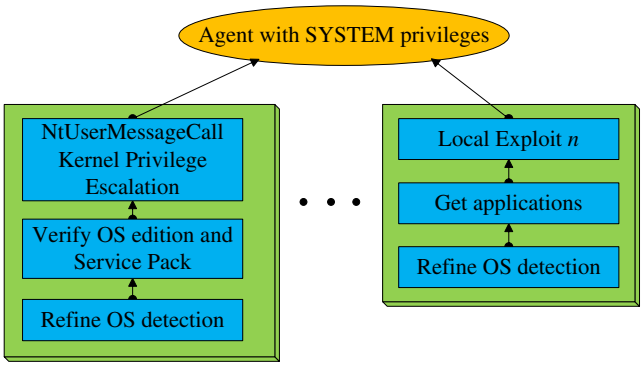

Figure 3: Multiple strategies for a Local Privilege Escalation.

For example consider an attacker who has installed an agent with low privileges on a host H running Windows XP, and whose goal is to obtain SYSTEM privileges on that host. The attacker has a set of n predefined strategies to perform this privilege escalation (see Fig. 3). An example of a strategy is: refine knowledge of the OS version; verify that the edition is Home or Professional, with SP2 installed; get users and groups; then launch the local exploit *Microsoft NtUserMessageCall Kernel Privilege Escalation* that (ab)uses the vulnerability CVE-2008-1084. More generally:

Problem 2. Let $\mathfrak{g}$ be a fixed goal, and $\{G_1, \ldots, G_n\}$ a set of n strategies, where each strategy G_k is a group of ordered actions. For a strategy to be successful, all its actions must be successful. As in Problem 1, the task is to minimize the expected total cost.

In this problem, actions are executed sequentially, choosing at each step one action from one group, until the goal $\mathfrak{g}$ is obtained. Considering only one strategy G, we can calculate its expected cost and probability of success. Suppose the actions of G are $\{A_1, \ldots, A_n\}$ and are executed in that order. Then the expected cost (e.g. expected runtime) of the group G is

$$T_G = t_1 + p_1\, t_2 + p_1\, p_2\, t_3 + \ldots + p_1\, p_2 \ldots p_{n-1}\, t_n$$

and, since all the actions must be successful, the probability of success of the group is simply $P_G = p_1\, p_2 \ldots p_n$.

PROPOSITION 2. *A solution to this problem is to sort the strategies according to the coefficient T_G/P_G (smallest value first), and execute them in that order. For each strategy group, execute the actions until an action fails or all the actions are successful.*

PROOF. In this problem, an attack plan could involve choosing actions from different groups without completing

all the actions of each group. But it is clear that this cannot happen in an optimal plan.[5]

So an optimal attack plan consists in choosing a group and executing all the actions of that group. Since the actions of each group G are executed one after the other, they can be considered as a single action with probability P_G and expected time T_G. Using the *choose* primitive, it follows that groups should be ordered according to the coefficients T_G/P_G. □

Multiple Groups of Actions

We extend the previous problem to consider groups of actions bounded by an AND relation (all the actions of the group must be successful in order to obtain the result $\mathfrak{g}$), but where the order of the actions is not specified. The difference with Problem 2 is that now we must determine the order of execution within each group.

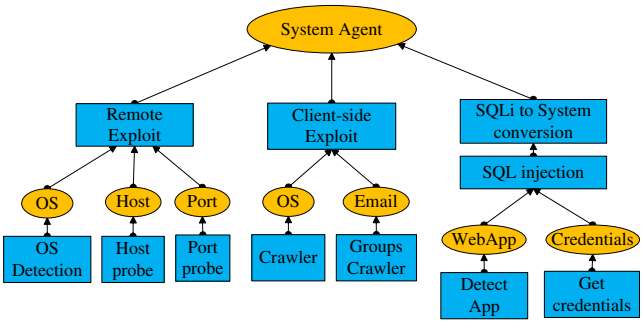

Figure 4: Probabilistic attack tree (with two layers).

Fig. 4 shows an example of this situation. A System Agent can be installed by using a Remote exploit, a Client-side exploit or a SQL injection in a web application. Each of these actions has requirements represented as assets, which can be fulfilled by the actions represented on the second layer. For example, before executing the Remote exploit, the attacker must run a Host probe (to verify connectivity with the target host), Port probe (to verify that the target port of the exploit is open), and an OS Detection module (to verify the OS of the target host).

Problem 3. Same as Problem 2, except that we have n groups $\{G_1, \ldots, G_n\}$ of unordered actions. If all the actions in a group are successful, the group provides the result $\mathfrak{g}$.

PROPOSITION 3. *Let $G = \{A_1, \ldots, A_n\}$ be a group of actions bounded by an AND relation. To minimize the expected total cost, the actions must be ordered according to the coefficient $t_k/(1-p_k)$.*

[5]Suppose that there are only two groups G_A and G_B, whose actions are $\{A_1, \ldots, A_s\}$ and $\{B_1, \ldots, B_t\}$ respectively. Suppose that in the optimal plan A_s precedes B_t. Suppose also that the execution of an action $B_j \neq B_t$ precedes the execution of A_s. Executing B_j will not result in success (that requires executing B_t as well), and it will delay the execution of A_s by the expected running time of B_j. Thus to minimize the expected total running time, a better solution can be obtained by executing B_j after the execution of A_s. This contradiction shows that all the actions of G_B must be executed after A_s in an optimal solution. This argument can be easily extended to any number of groups.

PROOF. If the actions are executed in the order $A_1, \ldots, A_n$, then the expected cost is

$$T_G = t_1 + p_1\, t_2 + \ldots + p_1\, p_2 \ldots p_{n-1}\, t_n \qquad (2)$$

This expression is very similar to equation (1). The only difference is that costs are multiplied by p_k instead of $\overline{p_k}$. So in this case, the optimal solution is to order the actions according to the coefficient $t_k/\overline{p_k} = t_k/(1-p_k)$. □

Intuitively the actions that have higher probability of failure have higher priority, since a failure ends the execution of the group. The coefficient $t_k/(1-p_k)$ represents a trade-off between cost (time) and probability of failure.

Wrapping up the previous results, to solve Problem 3, first order the actions in each group according to the coefficient $t/(1-p)$ in increasing order. Then calculate for each group G the values T_G and P_G. Order the groups according to the coefficient T_G/P_G, and select them in that order. For each group, execute the actions until an action fails or all the actions are successful.

We call it the *combine* primitive, because it tells you how to combine a group of actions and consider them (for planning purposes) as a single action with probability of success P_G and expected running time T_G.

5. USING THE PRIMITIVES IN AN ATTACK TREE

We apply below the *choose* and the *combine* primitives to a probabilistic attack tree, where the nodes are bounded by AND relations and OR relations. The tree is composed of two types of nodes, distributed in alternating layers of asset nodes and action nodes (see Fig. 5).

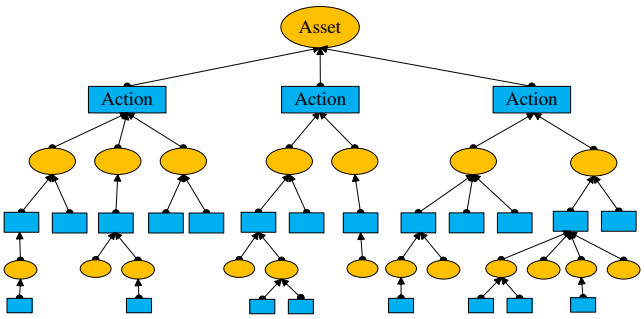

Figure 5: Attack tree with alternating layers of Assets and Actions.

An *asset node* is connected by an OR relation to all the actions that provide this asset: for example, an Agent asset is connected to the Exploit actions that may install an agent on the target host.

An *action node* is connected by an AND relation to its requirements: for example, the local exploit *Microsoft NtUserMessageCall Kernel Privilege Escalation* requires an agent asset (with low level privileges) on the target host H, and a Windows XP OS asset for H.

The proposed solution is obtained by composing the primitives from previous sections. In the AND-OR tree, the leaves that are bounded by an AND relation can be considered as a single node. In effect, using the *combine* primitive, that group G can be considered as a single action with compound probability of success P_G and execution time T_G.

The leaves that are bounded by an OR relation can also be (temporarily) considered as a single node. In effect, in an optimal solution, the node that minimizes the t/p coefficient will be executed first (using the *choose* primitive), and be considered as the cost of the group in a single step plan.

By iteratively reducing groups of nodes, we build a single path of execution that minimizes the expected cost. After executing a step of the plan, the costs may be modified and the shape of the graph may vary. Since the planning algorithm is very efficient, we can replan after each execution and build a new path of execution. We are assured that before each execution, the proposed attack plan is optimal given the current environment knowledge.

Constructing the Tree

We briefly describe how to construct a tree beginning with an agent asset (e.g. the objective is to install an agent on a fixed machine). Taking this goal as root of the tree, we recursively add the actions that can complete the assets that appear in the tree, and we add the assets required by each action.

To ensure that the result is a tree and not a DAG, we make an additional independence assumption: the assets required by each action are considered as independent (i.e. if an asset is required by two different actions, it will appear twice in the tree).

That way we obtain an AND-OR tree with alternating layers of asset nodes and action nodes (as the one in Fig. 5). The only actions added are Exploits, TCP/UDP Connectivity checks, and OS Detection modules. These actions don't have as requirements assets that have already appeared in the tree, in particular the tree only has one agent asset (the root node of the tree). So, by construction, we are assured that no loops will appear, and that the depth of the tree is very limited.

We construct the tree in this top-down fashion, and as we previously saw, we can solve it bottom-up to obtain as output the compound probability of success and the expected running time of obtaining the goal agent.

6. THE GRAPH OF DISTINGUISHED ASSETS

In this section we use the previous primitives to build an algorithm for *attack planning* in arbitrary networks, by making an additional assumption of independence between machines. First we distinguish a class of assets, namely the assets related with agents. We refer to them as *distinguished assets*. At the PDDL level, the predicates associated with the agents are considered as a separate class.

Planning is done in two different abstraction levels: in the **first level**, we evaluate the cost of compromising one *target* distinguished asset from one fixed *source* distinguished asset. More concretely, we compute the cost and probability of obtaining a target agent given a source agent. At this level, the attack plan must not involve a third agent. The algorithm at the first level is thus to construct the attack tree and compute an attack plan as described in Section 5.

At the **second level**, we build a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where the nodes are distinguished assets (in our scenario, the hosts in the target network where we may install agents), and the edges are labeled with the compound probability and expected time obtained at the first level. Given this graph, an initial asset $s \in \mathcal{V}$ (the local agent of the attacker) and a final asset $g \in \mathcal{V}$ (the goal of the attack), we now describe two algorithms to find a path that approximates the minimal expected time of obtaining the goal g.

The first algorithm is a modification of Floyd-Warshall's algorithm to find shortest paths in a weighted graph. Let $M = |\mathcal{V}|$ be the number of machines in the target network. By executing M^2 times the first level procedure, we obtain two functions: the first is $Prob(i, j)$ which returns the compound probability of obtaining node j from node i (without intermediary hops), or 0 if that is not possible in the target network; the second is $Time(i, j)$ which returns the expected time of obtaining node j directly from node i, or $+\infty$ if that is not possible. The procedure is described in Algorithm 1.

Algorithm 1 Modified Floyd-Warshall

```
P[i,j] ← Prob(i,j)   ∀ 1 ≤ i,j ≤ M
T[i,j] ← Time(i,j)   ∀ 1 ≤ i,j ≤ M
for k = 1 to M do
    for i = 1 to M do
        for j = 1 to M do
            T' ← T[i,k] + P[i,k] × T[k,j]
            P' ← P[i,k] × P[k,j]
            if T'/P' < T[i,j]/P[i,j] then
                T[i,j] ← T'
                P[i,j] ← P'
return ⟨T, P⟩
```

When the execution of this algorithm finishes, for each i, j the matrices contain the compound probability $P[i, j]$ and the expected time $T[i, j]$ of obtaining the node j starting from the node i. This holds in particular when $i = s$ (the source of the attack) and $j = g$ (the goal of the attack). The attack path is reconstructed just as in the classical Floyd-Warshall algorithm.

In a similar fashion, Dijkstra's shortest path algorithm can be modified to use the *choose* and *combine* primitives. See the description of Algorithm 2.

Algorithm 2 Modified Dijkstra's algorithm

```
T[s] = 0,  P[s] = 1
T[v] = +∞,  P[v] = 0   ∀v ∈ V, v ≠ s
S ← ∅
Q ← V (where Q is a priority queue)
while Q ≠ ∅ do
    u ← arg min_{x∈Q} T[x]/P[x]
    Q ← Q\{u},  S ← S ∪ {u}
    for all v ∈ V\S adjacent to u do
        T' = T[u] + P[u] × Time(u,v)
        P' = P[u] × Prob(u,v)
        if T'/P' < T[v]/P[v] then
            T[v] ← T'
            P[v] ← P'
return ⟨T, P⟩
```

When execution finishes, the matrices contain the compound probability $P[v]$ and the expected time $T[v]$ of obtaining the node v starting from the node s. Using the modified Dijkstra's algorithm has the advantage that its complexity is

$\mathcal{O}(M^2)$ instead of $\mathcal{O}(M^3)$ for Floyd-Warshall. Let n be the number of actions that appear in the attach trees, this gives us that the complexity of the complete planning solution is $\mathcal{O}(M^2 \cdot n \log n + M^2) = \mathcal{O}(M^2 \cdot n \log n)$.

7. OUR IMPLEMENTATION

We have developed a proof-of-concept implementation of these ideas in the Python language. This planner takes as input a description of the scenario in the PPDDL language, an extension of PDDL for expressing probabilistic effects [28].

Our main objective was to build a probabilistic planner able to solve scenarios with 500 machines, which was the limit reached with classical (deterministic) planning solutions in [16]. Additionally we wanted to tame memory complexity, which was the limiting factor. The planner was integrated with the pentesting framework Core Impact, using the procedures previously developed for the work [16]. The architecture of this solution is described in Fig. 6.

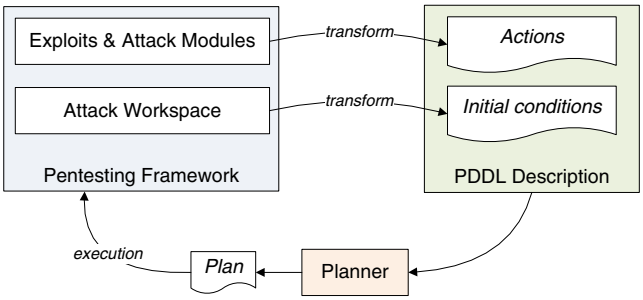

Figure 6: Architecture of our solution.

This planner solves the planning problem by breaking it into two levels as described in Section 6. On the higher level, a graph representation of *goal* objects is built. More concretely, there is a distinguished node for each host. The directed edges in this graph are obtained by carrying out the tree procedure described in Section 5, obtaining a value for the probability and the cost of obtaining the predicate represented by the target node, when the predicate represented by the source node is true.

The final plan can then be determined by using the modified versions of Dijkstra and Floyd-Warshall algorithms. The figures that follow show the planner running time using the modified Dijkstra's algorithm.

Testing and Performance

The experiments were run on a machine with an Intel Core2 Duo CPU at 2.4 GHz and 8 GB of RAM. We focused our performance evaluation on the number of machines M in the attacked network. We generated a network consisting of five subnets with varying number of machines, all joined to one main network to which the attacker initially has access.

Fig. 7 shows the memory consumption of this planning solution, which clearly grows linearly with M. Our current implementation manages to push the network size limit up to 1000 machines, and brings memory consumption under control.[6] For $M = 1000$, we are using less than 1 GB of RAM, with a planner completely written in Python (not optimized in terms of memory consumption).

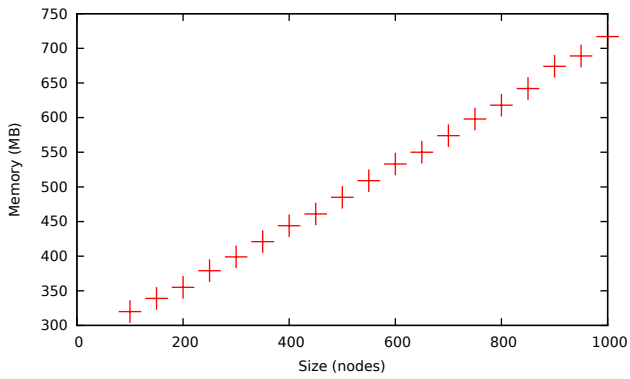

Figure 7: Memory consumption vs number of machines.

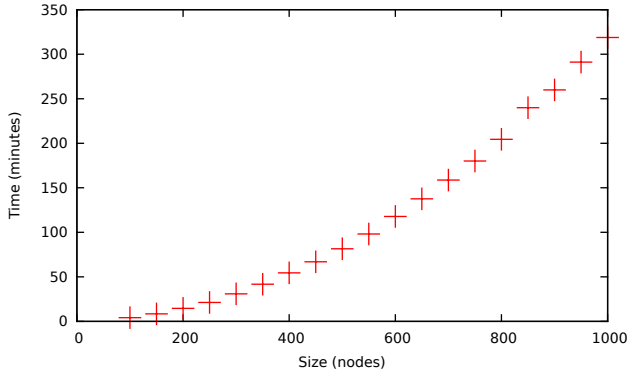

Figure 8: Solver runtime vs number of machines.

Fig. 8 shows the growth of solver running time, which seems clearly quadratic, whereas in [16] the growth was exponential. It should be noted however that, comparing only up to 500 machines, running times are slightly worse than those of the solution based on deterministic planners. This can be improved: since our planner is written in Python, a reasonable implementation in C of the more CPU intensive loops should allow us to lower significantly the running time.

And of course we added a notion of probability of success that wasn't present before. As a comparison, in another approach that accounts for the uncertainty about the attacker's actions [25], the authors use off-the-shelf solvers, managing to solve scenarios with up to 7 machines – and are thus still far from the network sizes reached here.

Both curves are compared in Fig. 9 showing the quadratic growth of solver runtime. In the testing scenarios, the nodes are fully connected, so we have to solve a quadratic number of attack trees. This figure also confirms in practice the computed complexity.

An interesting characteristic of the solution proposed is that it is inherently parallelizable. The main workload are the M^2 executions of the *first level* procedure of Section 6. This could be easily distributed between CPUs or GPUs to obtain a faster planner. Another possible improvement is to

[6] By contrast, in [16] the hard limit was memory: in scenarios with 500 machines we ran out of memory in a computer with 8 GB of RAM. The memory consumption growth was clearly exponential, for instance 400 machines used 4 GB of RAM. This was difficult to scale up.

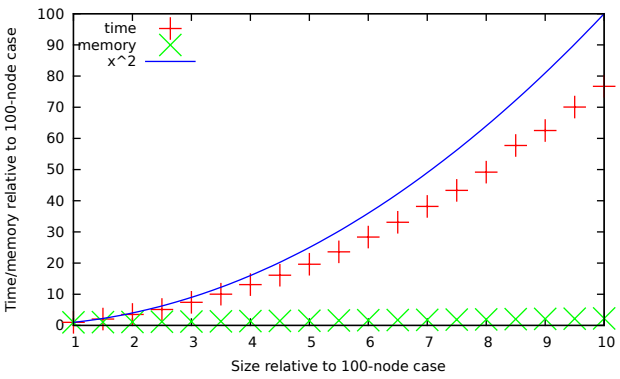

Figure 9: Time and memory relatives to the 100 machines case.

run the planner "in the cloud" with the possibility of adding processors on demand.

8. RELATED WORK

Early work on attack graph solving relied on model checking techniques [15, 27], with their inherent scalability restrictions; or on monotonicity assumptions [2, 18, 19] that are not able to express situations in which compromised resources are lost due to crashes, detection or other unforeseen circumstances.

The first application of planning techniques and PDDL solving for the security realm was [5], however this application was not focused on finding actual attack paths or driving penetration testing tools. In [12] attack paths are generated from PDDL description of networks, hosts and exploits, although the scenarios studied do not cover realistic scales. Previous work by the authors [16] addresses this limitation by solving scenarios with up 500 machines, and feeding the generated attack plans to guide a penetration testing tool. However, this work does not include probabilistic considerations. Recent work [9] also manages to provide attack paths to a penetration testing tool, in this case the Metasploit Framework, but again does not include probabilistic considerations.

Previous work by one of the authors [25] takes into account the uncertainty about the result of the attacker's actions. This POMDP-based model also accounts for the uncertainty about the target network, addressing information gathering as an integral part of the attack, and providing a comprehensive notion of attack planning under uncertainty. However, as previously stated, this solution does not scale to medium or large real-life networks.

9. SUMMARY AND FUTURE WORK

We have shown in this paper an extension of established *attack graphs* models, that incorporates probabilistic effects, and numerical effects (e.g. the expected running time of the actions). This model is more realistic than the deterministic setting, but introduces additional difficulties to the planning problem. We have demonstrated that under certain assumptions, an efficient algorithm exists that provides optimal attack plans with computational complexity $\mathcal{O}(n \log n)$, where n is the number of actions and assets in the case of an attack tree (between two fixed hosts), and $\mathcal{O}(M^2 \cdot n \log n)$ where M is the number of machines in the case of a network scenario.

Over the last years, the difficulties that arose in our research in *attack planning* were related to the exponential nature of planning algorithms (especially in the probabilistic setting), and our efforts were directed toward the aggregation of nodes and simplification of the graphs, in order to tame the size and complexity of the problem. Having a very efficient algorithm in our toolbox gives us a new direction of research: to refine the model, and break down the actions in smaller parts, without fear of producing an unsolvable problem.

A future step in this research is thus to analyze and divide the exploits into basic components. This separation gives a better probability distribution of the exploit execution. For example, the *Debian OpenSSL Predictable Random Number Generation Exploit* – which exploits the vulnerability CVE-2008-0166 reported by Luciano Bello – brute forces the 32,767 possible keys. Each brute forcing iteration can be considered as a basic action, and be inserted independently in the attack plan. Since the keys depend on the Process ID (PID), some keys are more probable than others.[7] So the planner can launch the *Debian OpenSSL PRNG* exploit, execute brute forcing iterations for the more probable keys, switch to others exploits and come to back to the Debian PRNG exploit if the others failed. This finer level of control over the exploit execution should produce significant gains in the total execution time of the attack.

Other research directions in which we are currently working are to consider actions with multidimensional numeric effects (e.g. to minimize the expected running time and generated network traffic *simultaneously*); and to extend the algorithm to solve probabilistic attack planning in Directed Acyclic Graphs (DAG) instead of trees. In this setting, an asset may influence the execution of several actions. This relaxes the independence assumption of Sections 4 and 5. Although finding a general algorithm that scales to the network sizes that we consider here seems a difficult task, we believe that efficient algorithms specifically designed for network attacks scenarios can be found.

Acknowledgments

Thanks to Ariel Futoransky and Ariel Waissbein for their contributions and insightful discussions.

10. REFERENCES

[1] D. Aitel. An introduction to MOSDEF. In *Black Hat Briefings, USA*, 2004.

[2] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224. ACM New York, NY, USA, 2002.

[3] I. Arce and G. McGraw. Why attacking systems is a good idea. *IEEE Computer Society - Security & Privacy Magazine*, 2(4), 2004.

[4] I. Arce and G. Richarte. State of the art security from an attacker's viewpoint. In *PacSec Conference*, Tokyo, Japan, 2003.

[7] The OpenSSL keys generated in vulnerable Debians only depend on the PID. Since Secure Shell usually generates the key in a new installation, PIDs between 2,000 and 5,000 are more probable than the others.

[5] M. S. Boddy, J. Gohde, T. Haigh, and S. A. Harp. Course of action generation for cyber security using classical planning. In *Proc. of ICAPS'05*, 2005.

[6] B. Burns, D. Killion, N. Beauchesne, E. Moret, J. Sobrier, M. Lynn, E. Markham, C. Iezzoni, P. Biondi, J. S. Granick, S. Manzuik, and P. Guersch. *Security Power Tools*. O'Reilly Media, 2007.

[7] Y. Chen, B. W. Wah, and C. Hsu. Temporal planning using subgoal partitioning and resolution in SGPlan. *J. of Artificial Intelligence Research*, 26:369, 2006.

[8] C. Domshlak and J. Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30(1):565–620, 2007.

[9] D. Elsbroek, D. Kohlsdorf, D. Menke, and L. Meyer. Fidius: Intelligent support for vulnerability testing. In *Working Notes for the 2011 IJCAI Workshop on Intelligent Security (SecArt)*, page 58, 2011.

[10] M. Fox and D. Long. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(2003):61–124, 2003.

[11] A. Futoransky, L. Notarfrancesco, G. Richarte, and C. Sarraute. Building computer network attacks. Technical report, CoreLabs, 2003.

[12] N. Ghosh and S. K. Ghosh. An intelligent technique for generating minimal attack graph. In *First Workshop on Intelligent Security (Security and Artificial Intelligence) (SecArt '09)*, 2009.

[13] J. Hoffmann. Extending FF to numerical state variables. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, pages 571–575, 2002.

[14] S. Jajodia, S. Noel, and B. O'Berry. Topological analysis of network attack vulnerability. *Managing Cyber Threats: Issues, Approaches and Challenges*, pages 248–266, 2005.

[15] S. Jha, O. Sheyner, and J. Wing. Two formal analyses of attack graphs. In *15th IEEE Computer Security Foundations Workshop, 2002. Proceedings*, pages 49–63, 2002.

[16] J. Lucangeli, C. Sarraute, and G. Richarte. Attack Planning in the Real World. In *Workshop on Intelligent Security (SecArt 2010)*, 2010.

[17] H. D. Moore. Penetration testing automation. In *SANS Penetration Testing Summit*, 2010.

[18] S. Noel, M. Elder, S. Jajodia, P. Kalapa, S. OŠHare, and K. Prole. Advances in Topological Vulnerability Analysis. In *Proceedings of the 2009 Cybersecurity Applications & Technology Conference for Homeland Security*, pages 124–129. IEEE Computer Society, 2009.

[19] S. Noel and S. Jajodia. Understanding complex network attack graphs through clustered adjacency matrices. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 160–169, 2005.

[20] C. A. Phillips and L. P. Swiler. A graph-based system for network-vulnerability analysis. In *Workshop on New Security Paradigms*, pages 71–79, 1998.

[21] M. Picorelli. Virtualization in software development and QA, 2006. WMWorld 2006.

[22] G. Richarte. Modern intrusion practices. In *Black Hat Briefings*, 2003.

[23] R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *IEEE Symposium on Security and Privacy*, pages 156–165. IEEE Computer Society, 2000.

[24] F. Russ and D. Tiscornia. Zombie 2.0. In *Hack.lu Conference*, Luxembourg, 2007.

[25] C. Sarraute, O. Buffet, and J. Hoffmann. Penetration testing == POMDP planning? In *SecArt'11*, 2011.

[26] C. Sarraute and A. Weil. Advances in automated attack planning. In *PacSec Conference*, Tokyo, Japan, 2008.

[27] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing. Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and Privacy*, pages 273–284. IEEE Computer Society, 2002.

[28] H. Younes and M. Littman. PPDDL 1.0: The language for the probabilistic part of IPC-4. In *Proc. International Planning Competition*, 2004.