Intelligent, Automated Red Team Emulation

Andy Applebaum, Doug Miller, Blake Strom, Chris Korban, and Ross Wolf
The MITRE Corporation
{aapplebaum, dpmiller, bstrom, ckorban, rwolf}@mitre.org

ABSTRACT

Red teams play a critical part in assessing the security of a network by actively probing it for weakness and vulnerabilities. Unlike penetration testing – which is typically focused on exploiting vulnerabilities – red teams assess the entire state of a network by emulating real adversaries, including their techniques, tactics, procedures, and goals. Unfortunately, deploying red teams is prohibitive: cost, repeatability, and expertise all make it difficult to consistently employ red team tests. We seek to solve this problem by creating a framework for *automated red team emulation*, focused on what the red team does post-compromise – i.e., after the perimeter has been breached. Here, our program acts as an automated and intelligent red team, actively moving through the target network to test for weaknesses and train defenders. At its core, our framework uses an automated planner designed to accurately reason about future plans in the face of the vast amount of uncertainty in red teaming scenarios. Our solution is custom-developed, built on a logical encoding of the cyber environment and adversary profiles, using techniques from classical planning, Markov decision processes, and Monte Carlo simulations. In this paper, we report on the development of our framework, focusing on our planning system. We have successfully validated our planner against other techniques via a custom simulation. Our tool itself has successfully been deployed to identify vulnerabilities and is currently used to train defending blue teams.

CCS Concepts

•**Security and privacy** → **Penetration testing;** *Logic and verification;* **Network security;** Intrusion detection systems; •**Computing methodologies** → **Planning under uncertainty;** Planning for deterministic actions; Modeling and simulation; •**Theory of computation** → Automated reasoning; Pre- and post-conditions;

Keywords

Formal methods in security; red teaming; penetration testing; automated planning, advanced persistent threat, network security

ACSAC '16, December 05 - 09, 2016, Los Angeles, CA, USA

ISBN 978-1-4503-4771-6/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2991079.2991111

1. INTRODUCTION

The importance of red team testing (red teaming) for modern enterprises cannot be understated. In these exercises, externally or internally sourced "red teams" – groups of security experts emulating attackers – attempt to test all aspects of an organization's security posture by launching repeated and complex attacks against the organization's enterprise computer network[1]. As a concept, red teaming stands opposite traditional computer network defense: instead of subjectively asking what the best way to defend a system is, red teaming provides a way to concretely measure whether a system is secure. In essence, red teams are designed to put network defenses *to the test*.

Red teaming moves beyond traditional penetration testing, a similar type of security audit focused on identifying and exploiting vulnerabilities in the network. Red teams, by contrast, conduct security assessments throughout the network, moving about as a real adversary would and, in essence, analyzing the system's resiliency in the face of an attacker that has broken through the perimeter. To gain the most of the exercise, organizations will typically have their defensive operations – referred to as the "blue team" – on standby, attempting to detect the red team as it moves through the network. Once complete, the red team's results are compared against the blue team's responses, with the network hardened based on the level of compromise the red team was able to effect.

Despite the benefits of red teaming, there are numerous hurdles that prevent organizations from employing it; factors such as cost, time, and personnel all play a significant role in the decision to conduct a red team exercise. Moreover, even if these logistical challenges are overcome, other difficulties can arise such as expertise and design. As an example, the red team members' level of training and knowledge will dictate what attacks and techniques they are able to execute against the network; if the red team members are not well trained, then the benefit of the red team exercise will be minimal. Designing a red team exercise and defining the domain of a test can be similarly challenging; topics that would need to be addressed include what hosts should be tested, what techniques and tactics are allowed, what the goal of the red team is, how the red team should report their results, etc. Effort spent establishing these design parameters adds to the complexity of executing a red team assessment. When taken together, these issues of cost, time, personnel, training, and design have created a muddied arena where the meaning and utility of red teaming can be extensively debated, making it difficult for organizations to incorporate red teaming into their internal security procedures.

[1]For this paper, we consider red teaming only in the context of testing enterprise computer networks, although we acknowledge the ubiquitous use of the phrase "red teaming" as it can apply to many other domains, cyber or not.

Contribution.

This paper presents our work on designing and implementing an *automated* red teaming system: the Cyber Adversary Language and Decision Engine for Red Team Automation (CALDERA). By creating an automated red teaming system, we immediately address the problems of cost, time, and personnel, as the system can conduct an assessment without requiring any operator involvement. Moreover, while the red team assessment runs autonomously, it can optionally be preconfigured by an operator to execute a *specific* type of test, allowing operators (i.e., blue teams or other organizational technical staff) to control the design of the test. Architecturally, CALDERA is highly extensible such that little overhead is needed in order to train it to execute new techniques.

As opposed to traditional penetration testing, our approach focuses on "post compromise" actions that an adversary can take within a network, leveraging the MITRE-developed framework Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK)[2] which taxonomizes common advanced persistent threat (APT) actions. The techniques, tactics, and procedures (TTPs) taken from ATT&CK drive the atomic actions that CALDERA is able to execute, with a custom planning system layered on top to guide intelligent decision making. CALDERA notably moves away from other automated red teaming and penetration testing solutions by providing a library of actions with capabilities more robust than just "exploit" and "scan." Internally, the system is hybrid and customizable, leveraging techniques from classical and conformant planning, Markov Decision processes, and Monte Carlo simulations. Initial results show that our tool has significant promise for future red teaming, outperforming other approaches in a custom simulation and successfully infecting live hosts in real-world testing.

The rest of this paper is organized as follows: Section 2 covers related work and provides background and context for our work; Section 3 provides the details of our framework, including a brief description of our infrastructure and an in-depth look at our implementation of the logic engine; Section 4 provides our results conducting a simulation of the logic engine; and Section 5 closes the paper, noting key areas for future work and briefly providing the details of our real-world implementation.

2. BACKGROUND

Defensive security and automation go hand-in-hand – modern enterprises largely rely on autonomous security software able to block and address threats without needing constant human supervision. Examples include firewalls, intrusion detection/prevention systems, network proxies, host anti-viruses, access control mechanisms, etc. While system administrators are still needed to configure, manage, and operate these suites, automation has taken much of the grunt work out, making the task of securely defending a network much easier. Automation here provides significant advances when assessing network security posture, measuring software patch levels, security controls, known threat indicators, and others with ease.

Still, automation of defensive tools is not enough. These approaches, while successful in identifying potential vulnerabilities, fail to consider the network's response in the face of a real-world cyber adversary. Attacks are not limited to vulnerability exploits: after the initial compromise, adversaries commonly leverage *intrinsic security dependencies* to strengthen and expand their footholds on a system. In these scenarios, select actions that the adversary executes may be benign by themselves – such as using internal Windows commands – but when combined or used maliciously, can cause significant damage. Adversaries often execute complex attacks that are missed by simple scanning tools; because adversaries do not always exploit vulnerabilities, their actions will often go unnoticed.

[2]https://attack.mitre.org/wiki/Main_Page

Automation for Offensive Security.

Attackers have used automation to employ malware in the form of worms and viruses. Well-known examples range from the destructive-and-obvious Internet Virus of 1988 [21] that brought the internet to a halt in 1988 to the destructive-yet-covert Stuxnet worm [12] that recently wreaked havoc on Iran's nuclear program. Both of these software suites ran completely autonomously, infecting their targets without the need of human controllers. However, in the context of red teaming, automated malware offers limited benefits: malware functionality is often a one-and-done, leveraging a specific vulnerability and then executing its payload. Malware logic is rigid and fixed, lacking the complexity needed to truly test a network, let alone pivot off of exploiting the security dependencies in a network. While certain automated malware may play a part in a red team's operation, the red team itself needs to be able to expose, in a precise and measurable way, exactly how a real adversary – not just a piece of code – could operate in the network.

Moving past traditional malware, automation has been used as an aid to red teams when launching exploits. Some tools in this space focus on specific exploits, while others provide an entire suite that can launch a variety of exploits. Here, the most commonly known and used framework is the popular penetration testing tool Metasploit [2]. Metasploit provides a simple console interface that operators can use to select, customize, and launch exploits. Additionally, Metasploit is highly extensible, allowing operators to chain exploits or scans together – for example, scripts can be used to scan a target for a vulnerability and then, if present, exploit it.

More interesting than automated exploit execution is the general automated red teaming problem – here, the task is not to execute a single exploit (interfacing with an operator or not), but rather to conduct a lengthy, in-depth analysis of the security of the network via real-world testing, exposing the intrinsic security dependencies of the target system. Early work in [15] sought to create a stepping-stone towards automation for red teams by utilizing Unified Modeling Language (UML) mock-ups alongside eXtensible Markup Language (XML) to create a model for red-teams to consult during a live test. Their model included information such as the attack's functionality, the attacker's profile, and the best way that the system should prevent the attack. Their system provides a solid foundation for red teams to consult, but, relying on free-text to convey information, was not sufficient for pure automation.

Perhaps the best known use of automation for red teaming comes from the use of attack graphs [3]. In this model, vulnerabilities in the system are identified and chained together to tackle the intrinsic security dependency problem: by chaining the logical outcomes of vulnerabilities together, defenders can build a bigger picture of their overall security posture. As a defender, employing attack graphs is straightforward – [14] combines the output of vulnerability scanners into a logical model to better diagram the ways that an attacker could harm the network.

From a red team perspective, attack graphs are also useful for coming up with plans of attack, albeit via manual creation [20]. Automating attack graph generation as a red team is much more difficult – the large amount of uncertainty, as well as the desire to not get caught, makes constructing such a graph very challenging. As a result, solutions to the automation problem have instead focused on the area of *planning* as the guidance for automation.

2.1 Automated Planning

Automated planning is a well-established field in the artificial intelligence community with numerous applications. In these scenarios, the challenge is to come up with a logical "plan" – a sequence of actions – to achieve a set of goals, all specified in a generic (typically predicate) logic. Perhaps the earliest – and best known – solution to this problem was the STRIPS [9] planning agent and framework published in 1971, which was able to find plans to achieve a given goal in a pre-defined scenario. Actions are encoded simply with their names, preconditions (what must be true to execute the action), and their effects (what is true after their execution). The planning program chains actions together to find the shortest path that achieves some goal state.

The automated planning research sphere has changed greatly since the original STRIPS paper. Since then, automated planning has blossomed to include multiple subproblems; the STRIPS case is an example of a *classical* offline planner, where each action is deterministic in its execution and outcome and the plan is precomputed before execution (online planners, by contrast, execute an action and then update their plan based on the results obtained by observing the system's response). Other categories of planners include *probabilistic* planners, where actions have probabilistic outcomes (e.g., Q is true after executing action a with probability .5) as well as partially-observable planners, where the planning agent has limited information about the environment that it is executing in (i.e., the planner has $E \subset \mathcal{E}$ as its environment).

Below, we outline related work using planning for automated red teaming, breaking down the work in this area into two sections based on observability – for a more in-depth study, readers should consult Hoffman's work in [10].

Strong Observability.

We categorize strong observability approaches as those that assume the defender's perspective with either full or near-full network knowledge. Our analysis starts with the work of Boddy et al. in [4]. Their approach featured similarities to traditional attack-graphs, though because they used a planning-based approach, their system could be encoded using more complex actions. Novel to their system was the ability to generate attacker plans for a variety of problem sets: their system was customizable based on predefined adversary characteristics, attack methods, network models, and adversary objectives. Given these inputs, their tool was able to generate potential courses of action that an adversary would take to reach their objective. While focused on the perspective of the defender (and therefore using fully-visible deterministic planning), their work is notable as it is one of the first to consider how an attacker could move through a network by using automated planning.

Planning has also been used explicitly over attack graphs. While these approaches tend to focus on the defender's perspective – using full knowledge of the network – they still have relevance in the automated red teaming domain. For example, [16] treats attack graphs as a traditional planning problem. They extend the traditional attack graph paradigm by encoding attacker and user profiles to identify more critical attack paths the adversary could take.

The approach in [13] uses a strict classical planning system for automated penetration testing. Unlike the models mentioned previously, their framework is embedded into a live exploit execution tool, in this case the proprietary Core Impact[3] suite. Their system includes a conversion from exploits enabled by the exploit execution tool into an open planning language; with this encoding, they are able to use standard planning software during execution. While their approach showed very promising results, it assumes a strong degree of observability, with little to no room for uncertainty.

Planning with Uncertainty.

This body of work is more applicable to real instances of red teaming, starting off with little or no information about the system(s) they are attacking. FIDIUS [8], for example, layers a planning system over the Metasploit framework as an exploit execution engine. Here, the tool uses *contingency planning* to incorporate sensing actions for services. In contingency mode, their planner constructs a branch for each possible if-then condition. While this model helps to handle uncertainty, it still requires explicit domain knowledge; in this case, FIDIUS still requires hosts, connections, and subnets to be pre-defined.

As opposed to traditional planning, [6] attempts to solve attack graphs with Markov Decision Processes (MDPs), which model the world as states and actions as transitions between states, with a reward function encoding the "reward" for moving from one state to another. Their work seeks to define an optimal policy – that is, what the best action is – for each state prior to execution based on using a fixed-lookahead horizon. By contrast, the approach in [11] features an *adaptive* attacker – i.e., using online techniques – alongside an MDP-solving system. Both of these MDP-based approaches assign probabilities to actions, moving uncertainty from the environment into uncertainty in the action's success. Note that these works focused on the *theoretical* problem of automated red teaming, lacking an implementation of their frameworks.

An extension to [13], the work in [19] brings uncertainty by using probabilistic planning: here, each action has a specific probability of success, which, like the MDP approaches, abstracts uncertainty about the environment into uncertainty that the action will succeed. Their planning system uses a custom planner – citing scalability issues when using off-the-shelf planning tools – with their own primitives. Their planner treats the problem more rigidly, structuring it akin to an attack graph problem. Like [13], this work also integrated with the Core Impact security tool.

Finally, the last category of automated red teaming that we cover is the work of Sarraute et al. in [18, 17]. Both of these works tackle the automated planning problem by using partially observable Markov decision processes (POMDPs) to encode the problem. As opposed to MDPs, POMDPs add a large amount of uncertainty to a system by encoding uncertainty regarding the *state* of the environment. In this scenario, the red team has a set of beliefs signifying how much it believes itself to be in a given state. Whether or not an action will be successful is a function in two parts: uncertainty in the action and uncertainty, from the point of view of the attacker, in the state of the environment.

The POMDP approach is perhaps the most comprehensive for automated red teaming – unlike attack graphs, traditional, contingency, or conformant planning, and unlike MDPs, it fully encodes all uncertainty in the environment. This is a very promising framework, and both [18] and [17] showed initial success. However, there were drawbacks with this approach, namely, in scalability: due to the complexity of POMDPs, solving a large problem is prohibitively large.

3. AUTOMATING RED TEAM DECISIONS

CALDERA works by combining two primary systems:

Virtual Red Team System (ViRTS) – the software infrastructure used to instantiate and emulate a red team adversary.

Logic for Advanced Virtual Adversaries (LAVA) – the logical

[3]https://www.coresecurity.com/core-impact-pro

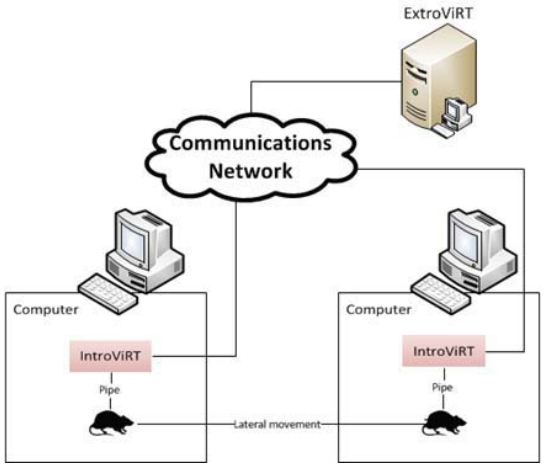

Figure 1: Visualization of CALDERA infrastructure.

model that CALDERA uses to choose which actions to execute.

The CALDERA infrastructure – instantiated by ViRTS – can be found in Figure 1. The system has two components: the master server (ExtroViRTS) and the remote access tool (RAT) clients on already infected hosts (IntroViRTS). Initially, CALDERA is configured such that exactly one enterprise host is infected with an IntroViRTS RAT and communication between that RAT and the master ExtroViRTS server is unencumbered. From this initial infection, CALDERA expands its foothold by using the LAVA engine to dictate what actions to take; the master server runs an instance of LAVA to select an action to execute, ultimately sending a command to a specific RAT in the field. That RAT executes the selected action, reporting all relevant details back to the master server. The master ExtroViRTS server updates its internal knowledge base, continuing to use LAVA to select future actions to be executed by the IntroViRTS clients.

This section provides an overview of the problem of automating red team decisions, focusing on the LAVA decision engine.

3.1 Post-Compromise Action Model

The works analyzed in Section 2 have all shown how automated planning can be used to viably construct an automated red team. However, these works all view red teaming through the lens of penetration testing: actions they encode are almost all categorized into either "exploit" or "scan for exploit." While this paradigm is suitable for vulnerability and penetration testing, it does not truly address the security dependency problem. Moreover, these exploit-and-scan systems lack the complexity of real adversaries and real red teams: in the real world, attacker TTPs go beyond exploit and scan.

Our approach is thus strongly motivated to consider the other side of exploits: what the adversary does *after* an exploit. This helps to better expose intrinsic security dependencies by creating an adversary model with many more capabilities than exploit execution. Moreover, we also want actions that are commonly executed, such that the automated red team system accurately emulates an adversary. We thus base our attack model on the MITRE-developed framework Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK)[4]. ATT&CK provides a common syntax to describe TTPs that an adversary can execute during the post-compromise attack phase. Notably, the entries in the ATT&CK taxonomy were largely informed by reports on real world activities engaged by APTs, in some cases corresponding directly to them.

[4]https://attack.mitre.org/wiki/Main_Page

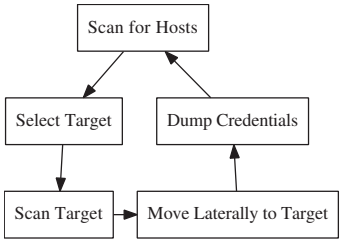

Figure 2: Simple finite-state machine used by early versions of CALDERA. Actions were executed as follows: the network was scanned for hosts, a target was selected, that target was scanned, the target was infected via lateral movement, credentials were dumped from the target, and then the process repeated.

The core of ATT&CK is a set of high level tactics that describe the goal of an adversary. These include lateral movement, exfiltration, persistence, and privilege escalation, among others. Within each tactic is a list of specific techniques that adversaries have used to achieve that tactic's goals. As an example, remote desktop protocol and pass the hash are techniques for lateral movement, and path interception and DLL injection are techniques for privilege escalation. CALDERA is designed to operationalize the ATT&CK model, containing modules that can implement select techniques – actions – when directed to.

3.2 LAVA: An Overview

The goal of the LAVA subsystem is to provide an intelligent way for CALDERA to select actions to be executed in order to simulate a red team. Initial iterations of CALDERA used a simple finite state machine: actions were encoded in a fixed order, including some minor conditionals, inside of the master ExtroViRTS server. Figure 2 gives a simple example where the ViRTS system ran on a fixed loop. While this system was effective, it suffered from multiple drawbacks:

- *Integrating new features.* When a new action was implemented in the ViRTS infrastructure, the underlying finite-state-machine needed to be manually reconfigured.
- *Predictability.* Due to its rigid internal logic, CALDERA showed a worm-like propagation through the network as opposed to a more fluid compromise typically seen in red teams and adversaries.
- *Customizability.* The finite-state-machine approach does not lend itself well to encoding profiles or preferences typically encountered in red teams and adversaries.

LAVA was designed to address these problems by replacing the finite-state-machine approach entirely, instead moving to a *modular* approach based on classical planning. Instead of executing actions in a pre-defined order, actions in LAVA are defined atomically with their logical requirements and effects upon execution. For decision making, LAVA strings actions together to create dynamic plans based on available knowledge and a given adversary profile. CALDERA consults LAVA for the best plan – once selected, the first action in that plan is executed by ViRTS, with subsequent plans updated as the system explores the network and updates the knowledge base.

$a \in \mathcal{A}$	$R(a)$
$dump\_credentials$	5
$lateral\_movement$	2
$exfiltrate\_data$	10

Action 1	Action 2	Action 3	$S(p)$
$exfiltrate\_data$	$dump\_credentials$	$lateral\_movement$	13.17
$exfiltrate\_data$	$lateral\_movement$	$dump\_credentials$	12.67
$dump\_credentials$	$exfiltrate\_data$	$lateral\_movement$	10.67
$dump\_credentials$	$lateral\_movement$	$exfiltrate\_data$	9.33
$lateral\_movement$	$exfiltrate\_data$	$dump\_credentials$	8.67
$lateral\_movement$	$dump\_credentials$	$exfiltrate\_data$	7.83

Figure 3: Example action scoring function (left) applied to a set of plans (right).

The next section discusses how this encoding enables actions to be selected by treating selection as a *planning* problem.

3.3 Automated Red Teaming as a Planning Problem

One of the biggest challenges that LAVA solves is what we term the *action selection problem*: how do we choose the best action in a given scenario to ultimately realize some pre-determined red teaming goal? In formal notation, we might say that given an environment $\mathcal{E}$ taken from some logical language $\mathcal{L}$, a set of actions $\mathcal{A}$, and some goal state $g \in \mathcal{L}$, how do we choose a single action $a \in \mathcal{A}$ that best realizes g? We believe the answer is to use *automated planning*.

3.3.1 Preliminaries

Automated penetration testing as a planning problem is extensively considered in [10], which provides a survey of approaches to simulated and automated pen-testing depending on the pen-testing scenario. They consider automated planning in the pen-testing sphere on two dimensions: the first being uncertainty, and the second concerned with how individual attack components interact with each other. They formalize these axes as follows: uncertainty in a given attack scenario can be none; can reside in the outcome of actions; or can be about the given state that the planner is in. Individual attack components can either take the form of an explicit network graph where actions only modify compromise; monotonic actions that have varying effects and cannot be undone; and generalized actions that can negatively impact each other. Plotted against each other, the authors recommend a specific formulation for each uncertainty versus interaction pair.

As our goal is to completely automate a red team assessment, we consider uncertainty from the point of view of *states*: by definition, the red team should have no knowledge of the underlying system it is trying to attack, including the system's and its own state. We assume actions in our model to be monotonic pieces: we consider explicit network graphs to be too rigid as red teams often engage in activities that do not directly result in compromise, and we believe that our approach can be scaled to general actions.

With these assumptions, [10] places this problem into the "Attack-Asset POMDP" problem sphere, suggesting a POMDP as the best underlying model. While this model has been shown to be successful in practice (discussed in Section 2), we believe that alternative approaches are viable. This belief stems from the following observation:

OBSERVATION 1. *Given complete knowledge of the environment, every action's execution would be completely deterministic: with full knowledge, an attacker would know both whether an action could be executed as well as the specific outcome(s) of that action.*

This is an important observation: the challenge in red teaming is not in the uncertainty of the outcome of an action, but rather uncertainty in the outcome *with regards to a given scenario*. If the planner had complete knowledge of the environment – that is, complete knowledge of the defender's system – it could construct plans using simple techniques such as STRIPS. Thus, with Observation 1 in mind, we treat uncertainty as *only* in the environment, labeling actions instead as deterministic and using a simple pre- and post-condition model similar to the requires/provides architecture as put forth in [22]. Formally, we define the actions as follows: let $\mathcal{A}$ be the set of actions that have been implemented in ViRTS for execution. For each action $a \in \mathcal{A}$, we define:

- a_{pre}: the set of preconditions that must be true in order to execute a.
- a_{post}: the set of postconditions that will be true upon executing a.

With this formulation, the challenge with plan creation in LAVA is two-fold: how are plans executed with regards to a specific goal, and how is uncertainty in the state of the world handled?

3.3.2 Executing Plans to Reach a Goal

Traditional STRIPS planning instances are given three inputs: the set of actions $\mathcal{A}$, the initial environment $\mathcal{E}$, and some goal predicate g. The goal of the planner is to construct a chain of actions, taken from $\mathcal{A}$, such that when executed in $\mathcal{E}$, their collective post-conditions will lead to g. While ideal for many planning problems, this setup is insufficient for red teaming for the following reasons:

- Often, though not always, red teams will have hard-to-encode goals, such as "gain as many footholds as possible" or "expose potential configuration vulnerabilities." These goals are doubly difficult to encode due to the inherent network and system uncertainty at the start of the test.
- In cases where there is a fixed, easy to encode goal (e.g., "obtain the credentials of a domain administrator"), the path to identifying that goal may be prohibitively large, or impossible to construct due to the attacker's uncertainty.

Both of these reasons have shifted our approach from planning offline to planning *online*. As opposed to offline planning, online planning does not necessarily attempt to construct a complete plan to reach a goal. Instead, online planners construct temporary plans that are modified during execution; after developing a plan, the planner executes the first action of the plan, observing the system's responses. If they are in line with what was expected, the planner continues, otherwise it creates a new plan.

Online planning is well suited to red teaming; the plan-as-you-go formulation makes it easier to adjust for uncertainty as the automated red team moves through the system. Moreover, it also helps to solve the goal-definition problem by allowing for *heuristics*. In particular, because the goal g may be poorly defined, we instead treat it as a *termination condition*, allowing an operator to

set heuristics to lead the planner towards that condition. Thus, our planning module is as follows:

1. Generate a list of plans, $\mathcal{P}$, given $\mathcal{A}, \mathcal{E}$, and g.
2. Assign each plan in $\mathcal{P}$ a score.
3. Select plan $p \in \mathcal{P}$ as the one with the best score.
4. Execute the first action, a, specified by p.
5. If terminating condition g is true, stop.
6. Otherwise, return back to step 1, updating $\mathcal{E}$ based on the post-conditions of a.

Note that this new online formulation is *flexible* for selection, as the heuristics can be tuned to either guide the planner long-term, or defined directly so that the planner explicitly works towards a specific goal. In our system, step 1 is generated by generating all plans of a fixed (user defined) depth.

Developing Heuristics.

Our algorithm uses a simple, customizable heuristic inspired by Markov decision processes (MDPs), the core of which is an *action-preference mapping*: here, the operator configures the function by assigning each individual action $a \in \mathcal{A}$ a numeric value – denoted $R(a)$ – signifying a "reward" for executing that action. Plans are then treated as sequences of actions (i.e., plan $p = \{a_1, a_2, ..., a_n\}$) and are then scored based on a decreasing-summation of the individual actions' scores. Put formally, given plan p, the score is defined as follows:

$$S(p) = \sum_{i=1}^{n} \frac{R(a_i)}{i}$$

Figure 3 shows an example scoring function applied to a set of plans. Note that the reward function R is highly customizable – in our case, we use it to map to tactics in the ATT&CK taxonomy, favoring certain tactics over others to simulate the way an adversary might realistically move through the network.

3.3.3 Planning Under Uncertainty

Our approach to handling uncertainty is tackle it from the point of view of *world-building*. This approach is inspired by [5]; in this paper, the authors tackle the problem of playing the chess variant Kriegspiel. In this adaptation, each opponent is unable to see the other's pieces, being instead given a set of incomplete observations dictated by a referee (e.g., the king is in check, a pawn can make a capture, a move is illegal). In order to construct plans, they use Monte Carlo simulations over the game board, guessing what the opponent's position might be (subject to prior observations as well as referee statements).

We take a similar approach to planning in LAVA. Let $E \subseteq \mathcal{E}$ denote the attacker's knowledge about the environment – for predicate $e \in E$, the attacker knows statement e is true. Planning using $\mathcal{A}$ and E is certainly possible, but long-term plans will lack depth due to the incompleteness of E. To help remedy this, LAVA features a world-simulator component which attempts to fill in unknown predicates $u \in \mathcal{E}$ with $u \notin E$. Formally, for action $a \in \mathcal{A}$, we define the *predicate space* P as follows:

$$P = \bigcup_{a \in \mathcal{A}} a_{pre}$$

To world build, we iterate over P, guessing which predicates are true based on currently available knowledge.

Figure 4 shows the entire operation of the planner. First, it creates a set of initial plans. Because these plans are incomplete – due to incompleteness in the knowledge base – the planner runs a set of simulations, selecting a specific action it believes is the best for each simulation. The simulations results are then tallied and the action that was selected the most times is chosen. After execution, the planner observes the responses, updating the internal model as well as the knowledge base based on the success of the action (which may fail if the simulations do not correlate with the real world $\mathcal{E}$) as well as the specific responses.

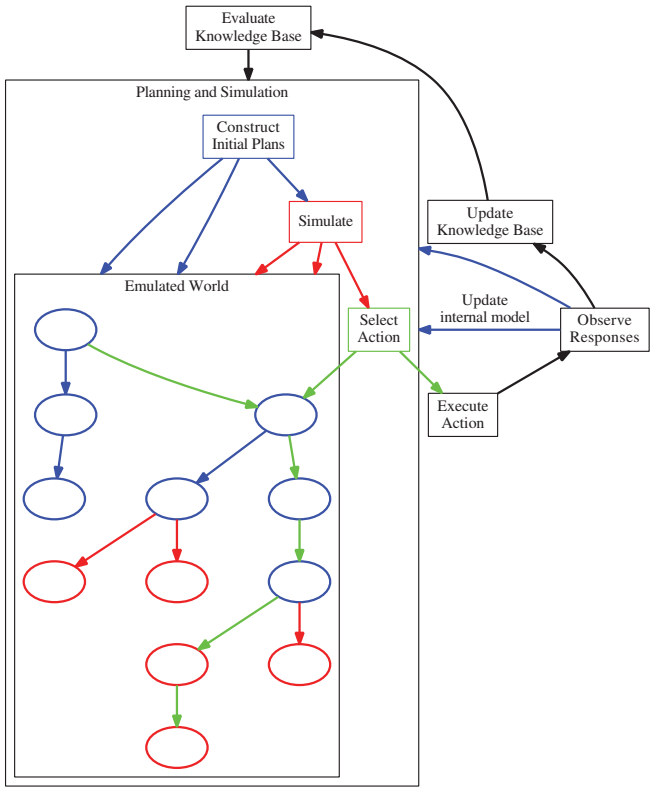

Figure 4: Visualization of internal LAVA operation. The planning engine first identifies an initial set of plans (blue), adding to them after executing a set of world-simulating procedures (red). After multiple simulations, an action is selected (green) and executed; responses in the real world are observed, with the knowledge base updated accordingly.

4. SIMULATED TESTING

We constructed a simulated environment to test the capabilities of the LAVA reasoning engine. Our simulation is built around a high-level representation of the ATT&CK model encoded into the $\mathcal{K}$ [7] planning language and implemented using the DLV$^{\text{K}}$ [1] planning system wrapped inside of `bash` scripts.

The goal of the simulation is to use a game-playing procedure to show how an adversary, using the LAVA framework, can attack the network. This intrinsically requires three components: a model for the "world" or system to be tested; the defender, who has a set of actions it can take and possesses near-complete knowledge of the network; and the attacker, who has a different set of actions and possesses a nearly-empty view of the network. Our simulation entails having the defender – representing the collective of all authorized users of the enterprise system – and attacker take turns until a terminating condition is met.

We define a generic template that both the attacker and defender inherit from. Formally, we write this as $\langle C_i, K_i, A_i \rangle$ where $i \in$

$\{\mathcal{A}, \mathcal{D}\}$ for the attacker and defender respectively. Each of these components matches the following definitions:

- C_i is the *state of compromise* and represents the current state of agent i, encoded as a set of logical predicates. Examples include system uptime for the defender and systems compromised for the attacker.
- K_i is the *state of knowledge* – a set of logical predicates denoting what the agent knows about the network. Both the attacker and defender have similar metrics for knowledge, including connections within the network, authorized remote logins, and the names of domain administrators.
- The last set, A_i, defined in our agent model is the set of actions – encoded with preconditions and effects – the particular agent can take. We refer to this as the agent's *capabilities*. Examples for the defender include taking a system offline and scanning a system while examples for the attacker include exploiting a vulnerability and dumping credentials.

In the remainder of this section we describe example cases for both the attacker and defender, focusing our efforts on defining the capabilities of the attacker and treating the defender as a largely passive agent in our system.

4.1 Modeling an Enterprise System

We simulate a networked Windows environment consisting primarily of workstations. Each workstation belongs to a set of domains, where each domain has at least one domain administrator. Similarly, each workstation has at least one local administrator and has a list of non-administrator accounts authorized for remote login. By default, local administrators and domain administrators are authorized for remote login. Workstations maintain bidirectional connections which specify which workstations can send network traffic to each other[5].

While not encoded logically, the internal world-construction module classifies workstations as either personal or shared – personal workstations have only one local administrator, one domain, and an empty or small list of authorized remote logins, while shared workstations can have multiple local administrators, multiple domains and many accounts authorized for remote login.

Figure 5 gives an example enterprise environment with three domains: `alpha`, `beta`, and `gamma`. In bold, we have the two shared servers `shar1` and `shar2` and six personal workstations `pers1` through `pers6`. Each workstation belongs to a set of domains, represented in purple. With each domain comes a set of domain administrators, represented in green. Local administrators are listed in blue, with non-administrator authorized remote logins in black. Note that the number of workstations, domains, connections, and other attributes generated by the world-construction module are dictated by a pre-defined configuration file.

The last component of the world is the concept of an active login, which denotes what accounts have recently logged into a workstation since the last reboot. In Figure 5, the set of recently logged in accounts is highlighted in italics. For example, `pers1` was not recently logged into, `pers2` was recently logged into by local administrator `roger` and domain administrator `john`, and `shar2` was recently logged into by remote account `tony`, local administrator `keith`, and domain administrators `john` and `ozzy`.

[5]In most enterprise systems, any workstation within a NAT can typically send traffic to any other workstation within the NAT. However, these traffic flows may be rare and use by an attacker could lead to easy detection. Our encoding of bidirectional connections might therefore be best understood as bidirectional connections that the adversary can use with impunity.

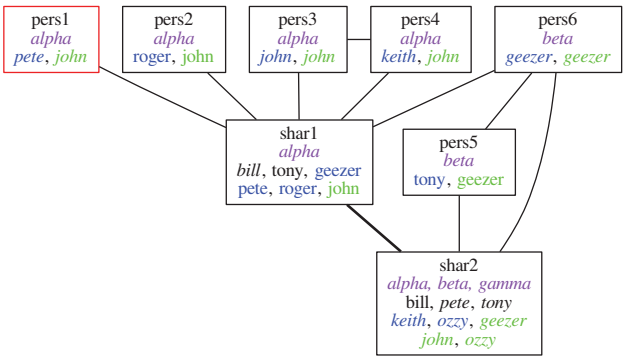

Figure 5: Example setup encoding of an enterprise system. Each box represents an individual workstation with edges representing allowed traffic flows. The name of the workstation is on the first line within each box, with the domains in purple, authorized remote logins in black, local administrators in blue, and domain administrators in green. Active logins are denoted by italics.

4.2 Defining the Defender

As our focus is on the attacker in this paper, we construct a simple, passive defender that is completely unaware of the adversary's presence, leaving the implementation of a full defender to future work. This defender maintains full and static view of the network state absent the details of the adversary's knowledge and state, yielding that $C_{\mathcal{D}} = \emptyset$ and $K_{\mathcal{D}}$ is full knowledge of the world. The defender can execute one of two defensive capabilities ($A_{\mathcal{D}}$):

Rebooting – This action signifies the case when the defender reboots a workstation. This can happen without any preconditions, and the consequence is it sets the number of active credentials stored on the system to zero.

Logging In – Here, the defender randomly chooses a workstation and eligible account (i.e., an account that is not currently logged in) and adds that account to the list of active credentials on that workstation.

With these actions, the defender operates completely randomly such that the decision to reboot or log in is dictated by the simulation configuration. Note that in this model, our defender is more of a system administrator due to its limited and naïve actions.

4.3 Defining the Attacker

Our example attacker emulates an APT with simple capabilities: credential dumping via Mimikatz, lateral movement using remote desktop, and exfiltration. These simple capabilities require a robust attacker model which we define below. Note that in all cases our attacker is *monotonic* – if some predicate $p \in \{C_{\mathcal{A}} \cup K_{\mathcal{A}}\}$ is true at turn t_i, then for all turns t_j with $t_i < t_j$, p is also true.

4.3.1 $C_{\mathcal{A}}$: State of Compromise

The adversary maintains six measures for its state of compromise:

- *Footholds*, which is the set of systems that the adversary has established a foothold on. By default, footholds are assumed to be persistent.

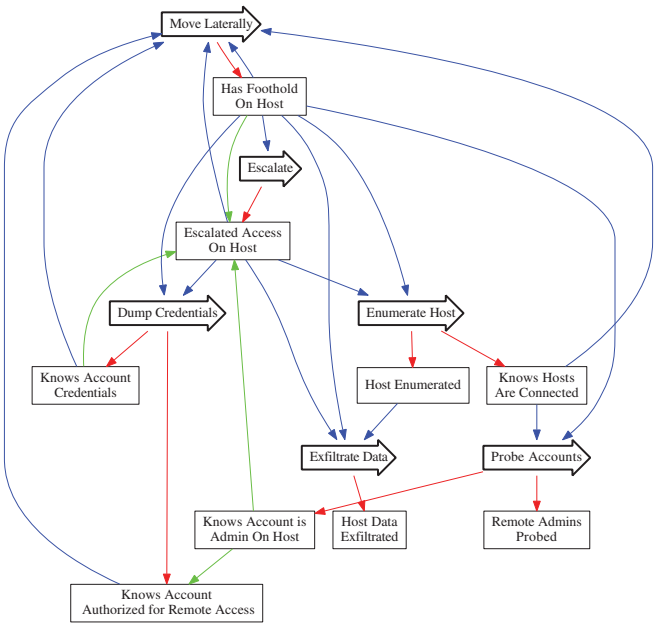

Figure 6: Dependencies between the actions in $A_{\mathcal{A}}$ and the states in $C_{\mathcal{A}}$ and $K_{\mathcal{A}}$. Actions are represented as bolded arrows and states are normal boxes. A blue edge from a state to an action signifies that that state is part of the *pre-conditions* for that action; a red edge from an action to a state means that that action modifies that state as part of its *post-conditions*. Green edges denote inferences that can be made between states (e.g., if the adversary knows an account is an admin on a host, then the adversary can infer that that account is authorized for remote access on that host).

- *Escalated footholds*, which is the set of systems that the adversary has execution access with elevated privilege on.
- *Known credentials*, which is the set of usable credentials that the adversary has obtained during the coarse of infiltrating the network.
- *Exfiltrated data*, which is the set of systems that the adversary has successfully exfiltrated all data from.
- *Remotely probed hosts*, which is the set of systems that the adversary has remotely probed for admins.
- *Hosts enumerated*, which is the set of systems that the adversary has performed local host enumeration on.

4.3.2 $K_{\mathcal{A}}$: State of Knowledge

Following the typical red team design, our attacker has *limited* knowledge about the system it is attacking. Therefore, we define the following metrics to denote its state of knowledge:

- *Known connections*, which is the set of valid network flows between workstations in the network.
- *Known authorized remote logins*, which denotes the accounts that are authorized to log in to a particular workstation on the network.
- *Known local admins*, which holds pairs mapping accounts to the workstations they are admins on.

For now, we do not encode adversarial knowledge about domains, treating domain admins as local admins.

4.3.3 $A_{\mathcal{A}}$: Capabilities

We define six attacker capabilities, each taking from a tactic and technique in ATT&CK.

- *Remote desktop protocol*. Here, the adversary remotely logs in from one workstation to another via remote desktop protocol (RDP). To do this, the adversary must know the credentials of an authorized remote login (admin or not) on the target computer, must know that there is a connection from the currently infected workstation to the target, must not currently have a foothold on the target, and must have elevated credentials on the currently infected machine. Once laterally moved, the attacker is said to have a foothold on the new machine. Using RDP is an instance of lateral movement in ATT&CK.
- *Dumping credentials with Mimikatz*. Here, the adversary gains access to all of the accounts recently logged into the machine that it is dumping the credentials on. The only preconditions for this capability is that the adversary must have an elevated foothold on the target machine. Credential dumping can be done even if it has been done before. This capability corresponds to a specific instance of credential access in ATT&CK.
- *Probe accounts*. The adversary targets another workstation on the network and obtains a list of local admins on that machine using a command such as NET USER. To do this, the adversary must have a foothold on another workstation on the network, must know that the target and infected workstations are connected[6], and must not have remotely probed accounts on the target prior. Relating to ATT&CK, this capability corresponds to the account discovery technique within the discovery tactic.
- *Enumerate local host*. The adversary gathers all useful information from a local host. This capability corresponds to both the discovery and collection tactics within ATT&CK. Within discovery, this capability is a catch-all for multiple techniques, including file and directory discovery, local network connections discovery, system information discovery, etc. Within collection, this capability corresponds to the data from local system technique. As a prerequisite to this capability, the attacker must have an escalated foothold on the infected machine and must not have already performed local host discovery. Consequently, the attacker will have learned all valid network flows to and from the currently infected machine and will be eligible to exfiltrate files from the infected machine.
- *Exfiltration*. Here, the adversary exfiltrates all useful data from an infected machine. To do this, the adversary must have an escalated foothold, must have already performed local host enumeration, and must not have already exfiltrated the data on the machine. Note that this capability corresponds to *both* exfiltration and command-and-control within ATT&CK, assuming the latter functionality is already setup.
- *Privilege escalation*. Finally, we add an open-ended ability for the attacker to escalate privileges on an infected machine, with the only prerequisite being that the attacker not have escalated permissions on the target. This capability is typically only executed once, as future lateral movement in our

[6]In certain situations, this precondition can be relaxed, such as if the target is a Domain Server.

#	Action	#	Action
1	escalate(pers1)	6	lateral(shar1, john)
2	enumerateHost(pers1)	7	dumpCreds(shar1)
3	probeAccounts(shar1)	8	enumerateHost(shar1)
4	exfiltrate(pers1)	9	probeAccounts(shar2)
5	dumpCreds(pers1)		

Table 1: A trace of the attacker's actions in an example simulation.

model will always give escalated access. Note that our open ended construction assumes that the adversary already has some specific instance of privilege escalation selected in its toolkit, e.g., path interception, exploitation of vulnerability, etc.

Figure 6 shows how the actions in $A_{\mathcal{A}}$ (arrows) logically relate – i.e., how their pre- and post-conditions relate – to the states in $C_{\mathcal{A}}$ and $K_{\mathcal{A}}$ (boxes).

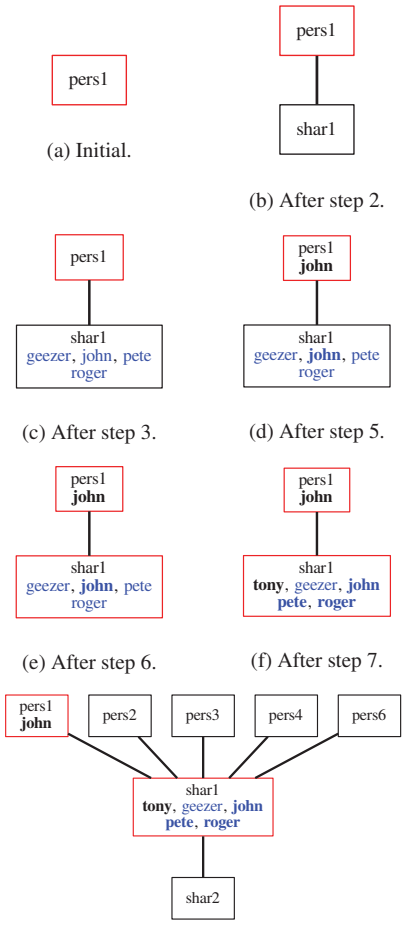

Figure 7: Attacker vision after the steps listed in Table 1.

4.4 Testing

We conducted a set of simulations to test the viability of our proposed planning mechanism. In these simulations, we ran four decision engines in four different scenarios, recording the number of adversary footholds on the network, credentials dumped by the adversary, and hosts exfiltrated at the end of the simulation. The scenarios ranged over the goal of the engine (exfiltration or lateral movement) as well as whether the network was randomly generated (with 10 personal workstations, 3 shared workstations, and 2 domains) or taken from a statically generated file (conforming to the network in Figure 5). Our results can be found in Table 3.

4.4.1 Decision Engines

We tested four different types of decision engines in our simulation. By design, actions other than credential dumping can only be executed once; this provides a small level of intelligence for the randomized engines we use, as they eventually choose "progressive" actions due to process of elimination.

- *Random*. During execution, an action is randomly selected from a list of legal options.
- *Greedy*. Similar to random – during execution, a list of legal moves is generated. If the goal action (i.e., exfiltration or lateral movement) is available, it is selected. Otherwise, an action is selected at random.
- *Finite-state Machine (FSM)*. Executes actions in a fixed order. If an action is to be executed but is not valid, it is skipped. Table 2 shows the order of actions.
- *Planner*. Uses the method described in Section 3.

Our expectation is that the planner will outperform all of the other engines. Additionally, we expect the greedy engine to outperform the random one – as it selects the goal action whenever it can – although we expect it to perform worse than the planner, and possibly equivalent to the FSM.

4.4.2 Setup

Each simulation was run as follows:

1. Initialize a network with the details specified in Section 4.1 either randomly or deterministically as the scenario dictates.
2. Allow the defender to execute an arbitrary number – set to 10 in this case – of reboot and login actions to seed the initial setup.
3. Place the adversary on the network with an initial foothold on some system.
4. Iterate as follows:
   (a) Allow the defender to move some arbitrary number of times (10).
   (b) Allow the attacker to pick an action and execute it, updating its knowledge base accordingly.
5. Stop the simulation after 20 attacker-defender move pairs.

An example trace of a game can be found in Table 1. Note that at each step, the attacker has *limited knowledge* as given in $K_{\mathcal{A}}$ – it is unable to see the full depth and inner workings of the network. Consequently, instead of seeing Figure 5, the attacker sees the network through a much more restricted lens. Figure 7 shows the attacker's limited vision after each of the moves in Table 1.

#	Action	#	Action
1	escalate	4	exfiltrate files
2	dump credentials	5	probe remote accounts
3	enumerate local host	6	move laterally

Table 2: Action order specified by the finite-state machine approach in Section 4.4.1. Note that the table loops – escalation follows lateral movement. During scenarios where lateral movement is the goal, exfiltration is completely removed from the finite-state machine.

4.5 Results

Each simulation was run fifteen times for each of the four engines in each of the four scenarios; Table 3 gives the average number of footholds, accounts, and exfiltrations each engine had over the five simulations. While our study has inherent limitations (e.g., limited trial size, model accuracy), our experiment led us to conclude the following:

- *The planner outperformed the other decision engines.* The planner scored nearly a whole host better in three of the scenarios. In the exfiltration over a fixed network scenario, the planner only slightly performed the next highest-scoring engine, however the planner on average laterally moved to more hosts in each simulation.
- *The random engine performed the worst, while the FSM and greedy engines performed roughly equally.* In each of the scenarios, the random engine scored the least. FSM and greedy both performed similarly in each scenario, although in the lateral movement over a random network the greedy engine on average moved to one more host than the FSM.
- *Credentials are not always important.* In three of the scenarios, the planner had the highest score with the lowest average number of credentials stolen. By contrast, the random engine typically had the most. The lack of credentials is due to the planner's approach; once it has sufficient credentials, it no longer looks for more. The random, greedy, and FSM engines all look for credentials at stages where it is not necessary, wasting time in doing so.

Scenario		Engine	Hosts	Accounts	Exfil.'d
Network	Goal				
Fixed	Exfil.	FSM	3.93	5.47	3.53
		Greedy	3.67	5.47	2.93
		Random	3.33	5.33	1.73
		Planner	4.47	3.93	4.13
Fixed	Lateral	FSM	4.87	5.6	
		Greedy	5.33	6.13	
		Random	3.73	9.07	
		Planner	6.33	5.07	
Random	Exfil.	FSM	3.13	8.23	2.53
		Greedy	3.13	7.6	2.53
		Random	3.27	8.8	1.6
		Planner	4.07	6.27	3.47
Random	Lateral	FSM	3.53	7.33	
		Greedy	4.73	11.47	
		Random	3.73	9.07	
		Planner	5.87	11.27	

Table 3: Simulation results for four different types of decision engines – one that chooses randomly, one built on a finite-state-machine, one on greedy action selection, and one that uses our planning method – over four simulation scenarios: with exfiltration on a fixed network, with exfiltration on a random network, without exfiltration on a fixed network, and without exfiltration on a random network. Results are measured in the number of footholds, known credentials, and hosts exfiltrated. Each scenario was simulated 15 times, with each simulation taking 20 turns and the defender making 10 moves per turn. Values represent the mean over the 15 trials.

5. CONCLUSIONS

This work showcased the architecture and design of our automated, intelligent red team emulation tool, CALDERA, combining the ViRTS execution infrastructure and the LAVA logical action model. Our approach provided a notable departure from prior work: first, our execution engine moved away from traditional automated adversary emulation by adding actions other than "exploit" and "scan," and second, our logical model featured a novel online planning approach custom-developed for red team emulation. The results in Section 4.4 show an improvement using our planning module over other, non-planning techniques, validating the intelligence component behind CALDERA.

Current Implementation.

At-present, capabilities corresponding to the host enumeration, credential access, and lateral movement tactics have been implemented in CALDERA, with each capability referencing a different technique. Of those implemented, the majority are executed by ViRTS by using standard Windows commands. Using LAVA, the ViRTS engine is able to string these actions together to create complex attack patterns, generating offensive artifacts as it moves through the network.

CALDERA has been fully tested over several small internal networks within the authors' organization. Early implementations have detected network misconfigurations and vulnerabilities. CALDERA is currently in use for active blue team training.

Future Work.

Future work on CALDERA will be in two forms: first, on the integration of new TTPs in the ViRTS infrastructure, and second, on theoretical changes to the LAVA module. Work on ViRTS will focus on providing a common format for TTPs so that operators can customize the specific instance of CALDERA that they would like to run, while also integrating with the logical model dictated by LAVA. From the theoretical side, we hope to implement and evaluate alternative approaches to the uncertainty problem, including incorporating standard planning languages and implementing and testing state-of-the-art planning solvers, i.e., MDP- and POMDP-based solvers. We also hope to expand on our theoretical simulation framework to allow for a more complex model and introduce more advanced evaluation metrics. With regards to the CALDERA software suite itself, we plan to run and test CALDERA within larger infrastructures, modifying our planning approach to account for bigger networks as needed.

6. REFERENCES

[1] K Planning System. http://www.dlvsystem.com/k-planning-system/.

[2] Metasploit penetration testing software. https://www.metasploit.com/.

[3] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In

Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS '02, pages 217–224, New York, NY, USA, 2002. ACM.

[4] Mark S Boddy, Johnathan Gohde, Thomas Haigh, and Steven A Harp. Course of action generation for cyber security using classical planning. In *ICAPS*, pages 12–21, 2005.

[5] Paolo Ciancarini and Gian Piero Favini. Monte carlo tree search in kriegspiel. *Artificial Intelligence*, 174(11):670 – 684, 2010.

[6] Karel Durkota. Computing optimal policies for attack graphs with action failures and costs. In *STAIRS*, pages 101–110, 2014.

[7] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. *Computational Logic — CL 2000: First International Conference London, UK, July 24–28, 2000 Proceedings*, chapter Planning under Incomplete Knowledge, pages 807–821. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

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

[9] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189 – 208, 1971.

[10] Joerg Hoffmann. Simulated penetration testing: From "dijkstra" to "turing test++", 2015.

[11] Leanid Krautsevich, Fabio Martinelli, and Artsiom Yautsiukhin. *Foundations and Practice of Security: 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers*, chapter Towards Modelling Adaptive Attacker's Behaviour, pages 357–364. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[12] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security Privacy*, 9(3):49–51, May 2011.

[13] Jorge Lucangeli Obes, Carlos Sarraute, and Gerardo Richarte. Attack planning in the real world. In *Working Notes for the 2010 AAAI Workshop on Intelligent Security (SecArt)*, page 10, 2010.

[14] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. Mulval: A logic-based network security analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[15] Helayne T. Ray, Raghunath Vemuri, and Hariprasad R. Kantubhukta. Toward an automated attack model for red teams. *IEEE Security and Privacy*, 3(4):18–25, July 2005.

[16] Mark Roberts, Adele Howe, Indrajit Ray, Malgorzata Urbanska, Zinta S Byrne, and Janet M Weidert. Personalized vulnerability analysis through automated planning. In *Working Notes for the 2011 IJCAI Workshop on Intelligent Security (SecArt)*, page 50, 2011.

[17] Carlos Sarraute, Olivier Buffet, and Joerg Hoffmann. Pomdps make better hackers: Accounting for uncertainty in penetration testing. In *Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI-12)*, 2012.

[18] Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. Penetration testing== pomdp solving? In *Working Notes for the 2011 IJCAI Workshop on Intelligent Security (SecArt)*, page 66, 2011.

[19] Carlos Sarraute, Gerardo Richarte, and Jorge Lucángeli Obes. An algorithm to find optimal attack paths in nondeterministic scenarios. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 71–80. ACM, 2011.

[20] Oleg Sheyner and Jeannette Wing. *Formal Methods for Components and Objects: Second International Symposium, FMCO 2003, Leiden, The Netherlands, November 4-7, 2003. Revised Lectures*, chapter Tools for Generating and Analyzing Attack Graphs, pages 344–371. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[21] Eugene H. Spafford. The internet worm program: An analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1):17–57, January 1989.

[22] Steven J. Templeton and Karl Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 Workshop on New Security Paradigms*, NSPW '00, pages 31–38, New York, NY, USA, 2000. ACM.