Alpaca: Building Dynamic Cyber Ranges with Procedurally-Generated Vulnerability Lattices

Joshua Eckroth, Kim Chen, Heyley Gatewood, and Brandon Belna
Stetson University
DeLand, Florida, USA
{jeckroth,kchen,hgatewood,bbelna}@stetson.edu

ABSTRACT

Developing cyber ranges for cybersecurity penetration testing and capture-the-flag challenges is normally a time-consuming process. A good cyber range challenges practitioners to find obscure paths to break into a system. The cyber range should encourage a "graph thinking" mindset, in which the attacker approaches the challenge from a variety of directions that may involve multiple steps before escalating privileges and solving the challenge. However, developing cyber ranges usually requires significant time and effort, and the solutions of many pre-made cyber ranges have already been published. We have developed Alpaca, a system that generates complex cyber ranges according to user-specified constraints. Using an AI planning engine and a database of vulnerabilities and machine configurations, the system is able to generate "vulnerability lattices," that is, sequences of vulnerabilities and exploits that achieve a user-specified goal. Alpaca also generates working virtual machines that include the vulnerabilities in the lattice. Constraints may be specified to require Alpaca to generate cyber ranges with a minimum or maximum complexity or require that certain vulnerabilities must be used to exploit the cyber range.

CCS CONCEPTS

• **Security and privacy**; • **Applied computing** → **Interactive learning environments**; • **Computing methodologies** → *Planning and scheduling*;

KEYWORDS

cyber range, penetration testing, cybersecurity

ACM Reference Format:
Joshua Eckroth, Kim Chen, Heyley Gatewood, and Brandon Belna. 2019. Alpaca: Building Dynamic Cyber Ranges with Procedurally-Generated Vulnerability Lattices. In *2019 ACM Southeast Conference (ACMSE 2019), April 18–20, 2019, Kennesaw, GA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3299815.3314438

1 INTRODUCTION

As the demand for cybersecurity professionals increases, so too does the demand for cybersecurity education. An essential part of cybersecurity education is practice with defending and exploiting systems. Secure systems depend on there being no path into a system, and insecure systems often have multiple paths an attacker can take. Successful attacks usually involve multiple steps, and it is important that we teach students to be able to think in this manner so they can better discover (and therefore protect) a vulnerable entry point. For example, say an attacker wants to get shell access on a web server, and their initial point of contact with the system is a web login form. From this simple login form, the attacker might be able to perform a SQL-injection to query the database for user account information and retrieve password hashes. Now that they have these hashes, they might be able to crack the hash for a specific user and use this information to log in to the web site. With this escalated privilege, they may next be able to exploit a deserialization vulnerability in the website to obtain shell access to the web server. And so on.

This "graph thinking" mindset is essential for defending systems. Dedicated attackers will eventually find any roundabout path into a system, often through seemingly innocuous entry points. Cybersecurity students may best acquire a similar mindset by practicing these kinds of circuitous exploits on live cyber ranges. However, sophisticated cyber ranges are difficult to build without some kind of automation assistance. Cyber ranges available to the public, such as those found on VulnHub,[1] are often already solved and therefore not appropriate for some educational scenarios such as exams. Furthermore, existing cyber ranges might not contain the vulnerabilities an educator wishes to emphasize. In this sense, these published cyber ranges are not dynamic, i.e., not responsive to an educator's particular goals.

We have developed Alpaca, a new open source tool[2] for building cyber ranges according to user-specified constraints. Unlike existing frameworks for building cyber ranges, Alpaca uses a vulnerability database and a custom planning engine to simulate sequences of exploits that would allow a skilled attacker to achieve a specified goal, such as root access. All paths that meet optional user-specified constraints, such as minimum number of steps or required use of a particular vulnerability, are found by Alpaca and collected into a vulnerability lattice. The lattice shows all possible ways to exploit the system. Once the lattice is found, a cyber range is built by automatically generating scripts to instantiate a virtual machine that contains all vulnerabilities that make up the lattice. Because Alpaca generates a vulnerability lattice before building the cyber range, the lattice serves as the solution for the cyber range. Naturally, a teacher would hide the lattice and just provide students with the cyber range.

ACMSE 2019, April 18–20, 2019, Kennesaw, GA, USA

ACM ISBN 978-1-4503-6251-1/19/04…$15.00
https://doi.org/10.1145/3299815.3314438

[1] https://www.vulnhub.com
[2] https://github.com/StetsonMathCS/alpaca

The rest of this paper is organized as follows. In the next section, we examine related work in generating cyber ranges. Section 3 explains in more detail our notion of a vulnerability lattice. Section 4 extensively covers each step in Alpaca's methodology for building dynamic cyber ranges. Next, Section 5 identifies some use cases and Section 6 gives a preliminary evaluation of Alpaca's implementation. Then we discuss in Section 7 numerous next steps for future work, and conclude in Section 8.

2 RELATED WORK

Several researchers have addressed the need for auto-generated cyber ranges. We will review several recent frameworks and explain how Alpaca differs and adds significant value in its unique approach.

SecGen [5] is a framework for generating randomized virtual environments for cybersecurity education and capture the flag (CTF) competitions. Cyber ranges are built through a random selection of modules. A SecGen module defines the base OS platform, vulnerabilities, services, utilities, networks, generators, and/or encoders. When given specific constraints, SecGen randomly selects a module that fits the required specifications, which are converted into Vagrant and Puppet scripts for instantiating the cyber range. Data generation and encoding modules may be nested to support random CTF flags. For example, a random textual flag may be generated and then encoded by a function such as Base64 encoding.

SecGen shares many similarities with Alpaca. Using a database of vulnerabilities and other system configurations, which they call modules, a live cyber range may be built by an automated process that randomly selects modules, then generates and runs provisioning scripts, resulting in a virtual machine (or multiple virtual machines) that contain the vulnerabilities and flags. Likewise, Alpaca allows a user to specify some constraints and then proceeds to define configuration scripts that provision a virtual machine with certain vulnerabilities. SecGen and Alpaca differ significantly in one important respect. Since Alpaca's vulnerability database defines pre- and post-conditions for each vulnerability, Alpaca is able to use a custom planning engine to generate a sequence of exploits that begin at a specified initial state (e.g., just a web login form), and end at a specified goal state (e.g., root access). There may be many ways to achieve the goal from the initial state, and Alpaca is able to find them all and generate cyber ranges that include these paths. A user may also provide additional constraints to Alpaca to require that, for example, no path from initial state to goal is shorter than N steps or that some particular vulnerability is required to reach the goal. Since SecGen does not use such a planning engine, it is not capable of building and reasoning about multi-step scenarios. Yet multi-step exploits are commonplace in real-world attacks so it is important that students practice with such scenarios.

CyTrONE [1] is a framework for generating cyber ranges for training scenarios. CyTrONE uses a simple configuration language to define the properties of the cyber range. The configuration is defined by a teacher using a web-based interface. CyTrONE generates both a training script with a series of questions and answers for the trainee, and a cyber range configuration. The authors include a training database that covers all of the techniques defined by the U.S. NIST Technical Guide to Information Security Testing and Assessment [4]. Cyber range instantiation is accomplished by CyRIS [3] using configuration scripts that define the software to be installed and network settings. CyRIS also has the ability to emulate attacks so that students can practice with traffic capture and analysis. Both CyTrONE and CyRIS lack the capability to automatically generate complex cyber ranges according to user-specified constraints. Rather, the user (teacher) must define the configuration of the cyber range, though CyTrONE's web-based interface makes this process relatively easy.

Burket et al. [2] developed an automatic problem generation tool for creating capture the flag scenarios. Their tool generates randomized variations of pre-defined templates for CTF scenarios. Since the cyber ranges in a CTF should pose relatively equal challenge to the teams, the scenario templates are defined ahead of time to have similar complexity. However, their tool is not capable of generating complex scenarios requiring a sequence of multiple exploits. Trickel et al. [6] also developed a means for generating cyber ranges for CTF competitions, but their system relies on a user-specified scenario configuration, i.e., a list of vulnerabilities to include in the virtual machines.

Unlike these related works, Alpaca focuses on generating complex, multi-step exploit scenarios in order to help students practice with "graph thinking" and thereby learn how to protect against attackers who have a similar mindset. At the core of Alpaca's approach is the concept of a vulnerability lattice, which identifies the steps an attacker may take to exploit a system.

3 VULNERABILITY LATTICES

A cyber range contains certain vulnerabilities that may be exploited to achieve certain goals. For example, a cyber range may contain a SQL-injection vulnerability and weak password hashes, thus enabling a student to obtain and crack the password hashes and achieve a privileged login. The same cyber range may support another way to achieve the same goal such as brute force login attempts or a deserialization and remote code execution attack, among other vulnerabilities. This suggests that there may be multiple paths, some sharing common intermediate steps, from the start state (cyber range instantiation) to a goal state (privileged login).

We may represent the variety of possible sequences of vulnerabilities required to achieve a particular goal as a lattice, since all such sequences start from the same initial state and end at the same goal state. Figure 1 shows an example of a vulnerability lattice.

We consider a cyber range to be an environment that "enables" a particular vulnerability lattice. This means that a skilled person who knows the full details of the corresponding vulnerability lattice for a particular cyber range would be able to exploit the cyber range, that is, reach the goal state in the lattice, following any of the paths specified in the lattice.

Generating a cyber range from a vulnerability lattice requires that each step in the lattice, such as SQL-injection, is realized in the cyber range in some concrete form. Each vulnerability in the database contains a "configuration" that indicates how it will be realized in a virtual machine. This configuration may specify certain versions of software, database structure and initial data, user

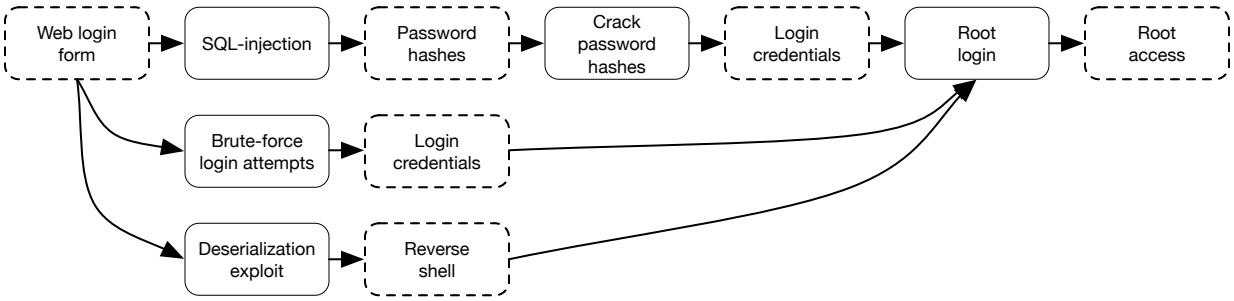

Figure 1: An Example of a Vulnerability Lattice. Dashed-line Boxes Represent States, Solid-line Boxes Represent Vulnerability Exploits

logins and passwords, and so on. As will be described below, Alpaca constructs scripts to generate the cyber range from a given lattice. For each vulnerability in the lattice, its configuration is converted into a script that "installs" or configures that vulnerability in the cyber range. For example, a SQL-injection vulnerability will produce a script that copies a pre-coded PHP file with a web login form, installs a database, and creates and fills a table of hashed passwords. Other vulnerabilities, like cracking password hashes, have an empty configuration and no corresponding script as no particular software or system configuration is needed on the cyber range for a penetration tester to be able to perform this exploit.

3.1 Measuring Complexity

Alpaca will find all paths that allow an attacker to start at the initial state and, through a sequence of exploits, arrive at the goal state. However, a teacher or capture-the-flag competition organizer may wish to prefer easier or harder scenarios depending on their goals. Thus, we developed a measure of complexity for a vulnerability lattice. The teacher may then define constraints that require the generated lattices have a complexity within a certain preferred range.

Generally speaking, a cyber range is more difficult to exploit if there are fewer distinct paths to exploitation and if some vulnerabilities in those paths are more difficult to exploit. On the other hand, the presence of more alternative paths makes the cyber range easier. Given a lattice that represents the possible paths for exploitation, we can define a measure of complexity that meets this general description.

Specifically, our desiderata for the complexity measure are as follows: 1) if a lattice A has more paths than lattice B, its complexity is lower and 2) if a lattice A has longer paths than lattice B, its complexity is higher. A formula that meets these desiderata is as follows,

$$C = \frac{1}{\sum_{\text{paths}}(1/\sum_{\text{weights}} w)},$$

where *paths* is the set of distinct paths through the lattice and *weights* are the individual weights on each vulnerability in the path. A vulnerability's weight is a subjective indication of its difficulty. A higher weight means the vulnerability is more difficult to exploit. If vulnerabilities are not assigned weights, then all weights are assumed to equal 1, so $1/\sum_{\text{weights}} w$ is effectively 1/path length. The complexities of various lattices are shown in Figure 2.

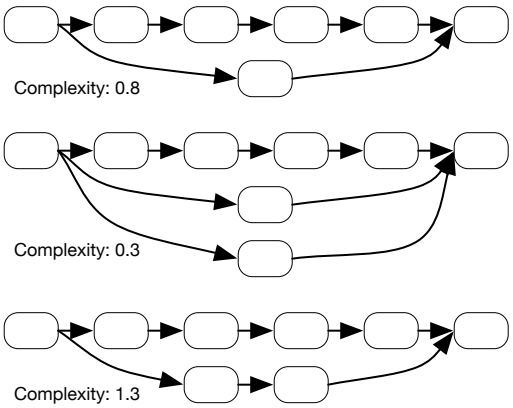

Figure 2: Examples of the Vulnerability Lattice Complexity Measure. For this Illustration, each Vulnerability is Assumed to have Weight (i.e., Difficulty of Exploitation) of 1.0

4 METHODOLOGY

Generating cyber ranges according to specified criteria requires a number of inventions. Our contribution to the cybersecurity community is the realization of these inventions in Alpaca. They may be summarized as follows.

(1) Development of a vulnerability database. Each vulnerability includes pre-conditions, post-conditions, a configuration, and a weight.
(2) An efficient planning engine that is capable of sequencing vulnerabilities to reach the specified end goal from the specified initial state.
(3) A means of "merging" configurations so that the vulnerabilities in a path have compatible configurations; a "path-configuration" is generated by merging the individual vulnerability configurations.
(4) A procedure for finding paths whose path-configurations are compatible, and grouping these compatible paths together to generate a vulnerability lattice and a "lattice-configuration."
(5) A means of generating scripts from a lattice (technically, from its lattice-configuration) to generate a cyber range that supports the vulnerabilities specified in that lattice.

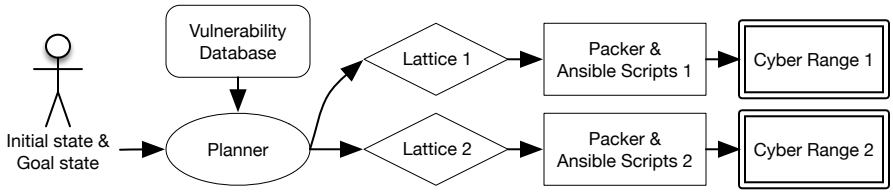

Figure 3: High-level Workflow of Alpaca. A User Specifies the Initial and Goal States, which the Planner Uses in Combination with the Vulnerability Database to Generate One or More Lattices. Each Lattice May then be Turned Into a Cyber Range Via Auto-generated Packer and Ansible Scripts

Each of these features will be described in more detail below.

These features work together in a workflow represented in Figure 3. First, a user specifies the initial state and goal state. These states take the form of lists of state properties as found in a vulnerability's pre-conditions and/or post-conditions. For example, possible state properties include ssh, login_page, hashed_passwords, and so on. Given the initial state and goal state, the planning engine then consults the vulnerability database to construct all possible paths (sequences of vulnerabilities) that start at the initial state and end at the goal state. Next, compatible subsets of these paths are found. Two paths are compatible if their path-configurations can be merged. These compatible subsets form lattices, in which all paths in a lattice are compatible in terms of their path-configurations, and the merge of these path-configurations is known as the lattice-configuration. Each lattice then may be instantiated as a cyber range using auto-generated scripts that reflect the lattice-configuration. The user may choose to instantiate one or more of the lattices as separate cyber ranges.

4.1 Vulnerability Database

The vulnerability database consists of structures that state the vulnerability's identifier, e.g., sql-injection or crack-hashes, its pre-conditions and post-conditions as lists of state properties, e.g., [login_page], its configuration, and its weight. A configuration is a key-value map. The keys refer to particular software, like 'php' or 'ssh.' The values give the required configuration for that software as another key-value map. In this inner map, the keys are arbitrary but often refer to the software version, or required system users and passwords, or files installed in the system. The values use the prefix "exists" or "only" (explained below), and are followed by either a list of values or a function that is able to generate random values, such as passwords. Below is an example vulnerability in the database.

```
vuln('sql-injection',
     % pre-conditions
     [login_page],
     % post-conditions
     [database_queries],
     % required configuration (key-value pairs)
     [apache-[modules-(exists, ["php"])],
      php-[version-(only, ["5.4.0"])],
      git-[clone-(exists, ["https://...", "/var/www/demo"])],
      mysql-[tables-(exists, ["demo.users", "hashes.sql"]),
           mysql_root_password-(only, [generatePassword])]]).
```

Configurations are used for two purposes. First, they provide information for generating scripts that instantiate the cyber range to support the vulnerability. For example, PHP may need to be installed with a certain version and certain PHP files may need to be cloned from a Git repository and placed in a certain directory. The second purpose of vulnerability configurations is to ensure the vulnerabilities in a cyber range are compatible with each other. The vulnerabilities must be able to co-exist. If two vulnerabilities require two different versions of PHP, or different MySQL root passwords, for example, the two vulnerabilities cannot coexist in the same cyber range. We need to ensure that we never generate a path or lattice with vulnerabilities that have incompatible configurations.

Two configurations are compatible if and only if:

- They contain distinct keys.
- Or, for all keys in common, consider the values (which are also key-value pairs). These values are compatible if:
  - They contain distinct keys.
  - For all keys in common, every value must be compatible. Values are written as `(exists, List)` or `(only, List)`. Consider two values, `(X1, List1)` and `(X2, List2)`. These values are compatible if `X1=X2=exists` or if `X1=only` and `List1`$\cup$`List2=List1`.

For example, configurations `[ssh-[users-(exists, ["jane", "john"])]]` and `[ssh-[users-(exists, ["alice"])], php-[version-(only, ["5.4.0"])]]` are compatible, but `[ssh-[users-(only, ["jane"])]]` and `[ssh-[users-(exists, ["alice"])]]` are not.

Some times, we wish to generate rather than hard-code configuration values, such as random passwords. Rather than providing a specific value in the vulnerability's configuration, we specify a function that can generate the value. In the example above, we included a configuration with generatePassword rather than a hard-coded password. This allows us to produce cyber ranges with equivalent vulnerability lattices but requiring different solutions, thus ensuring that students must go through all the steps to solve their individual cyber ranges.

Alpaca's vulnerability database contains 21 entries so far. These are shown (without configurations to save space) in Table 1 and graphically, connected by pre- and post-conditions, in Figure 4.

Creating new vulnerabilities is a relatively straightforward process. The vulnerability must be assigned a name and given pre-conditions and post-conditions that allow it to link up with other vulnerabilities that produce or consume those pre-/post-conditions. The vulnerability's configuration must also be specified. Specifying the configuration requires considering the other vulnerabilities in the database, since other vulnerabilities may restrict a software's

configuration in a way that may impact the new vulnerability. For example, if the new vulnerability requires an OpenSSH server, its configuration may simply state, `[openssh-[]]`, meaning that an OpenSSH server must be installed but no additional details are specified. However, if a different vulnerability states in its configuration, `[openssh-[port-(only, [2222])]]`, then the two configurations will be considered compatible, yet the SSH server is running on a non-standard port. Thus, some finesse is required to specify configurations for all vulnerabilities in the database in such a way that they are not overly specific to be too laborious to specify but also sufficiently specific to avoid making assumptions that may be silently violated by other configurations.

During cyber range generation, the merged lattice configuration is converted into scripts that build a virtual machine. We use Packer, from HashiCorp,[3] to instantiate a VirtualBox virtual machine. We use the Ubuntu 18.04 Linux distribution for our virtual machines, though any other distribution would work as well. Packer actually initiates the install process from an Ubuntu CD installation file (.iso file), so in future work we are free to change the Linux distribution by just changing the script that Packer executes. We can also add support for Windows virtual machines.

Next, we use Ansible[4] to install and configure all the software specified in the lattice configuration. Using an Ansible script template, Alpaca automatically fills in details in the template based on the lattice configuration.

Finally, after running Ansible, Packer exports the configured virtual machine to a virtual machine appliance in Open Virtualization Format. This file can be provided to students who can import the virtual machine into VirtualBox, VMWare, etc.

Further details of how lattice configurations are converted into Ansible scripts are provided in Section 4.5.

4.2 Planning Engine

Given an initial state and a goal state, specified as lists of state properties, the planning engine then finds all possible paths from the initial to goal states. A path consists of a sequence of vulnerabilities. Each vulnerability in the path adds its post-condition state properties to the current state, and each vulnerability requires that the current state contains its pre-conditions (which may be empty).

This planning problem is a typical backwards-chaining problem in which we first find a vulnerability V1 that has the goal state in its post-conditions, then find a vulnerability V2 that has one or more of V1's pre-conditions in V2's post-conditions, etc. Note, in order to link V1 with V2 in the path, their configurations must be compatible. Ultimately, we produce all possible paths from the initial state to the goal state. These paths are later collected into one or more lattices, which are then used to generate cyber ranges.

The planning engine is implemented in Prolog. Thus, it is straightforward to add arbitrary constraints to restrict the paths that are generated. For example, a teacher may wish to ensure all paths include a certain vulnerability. Or, he or she may wish that the cyber range not include certain state properties, e.g., to avoid allowing simple exploits like the vsftpd 2.3.4 backdoor. Such a constraint may be appended to the call to generate all paths. Due to Prolog's natural backtracking mechanisms, if the constraint fails, a path will be abandoned and another generated until the constraint is met.

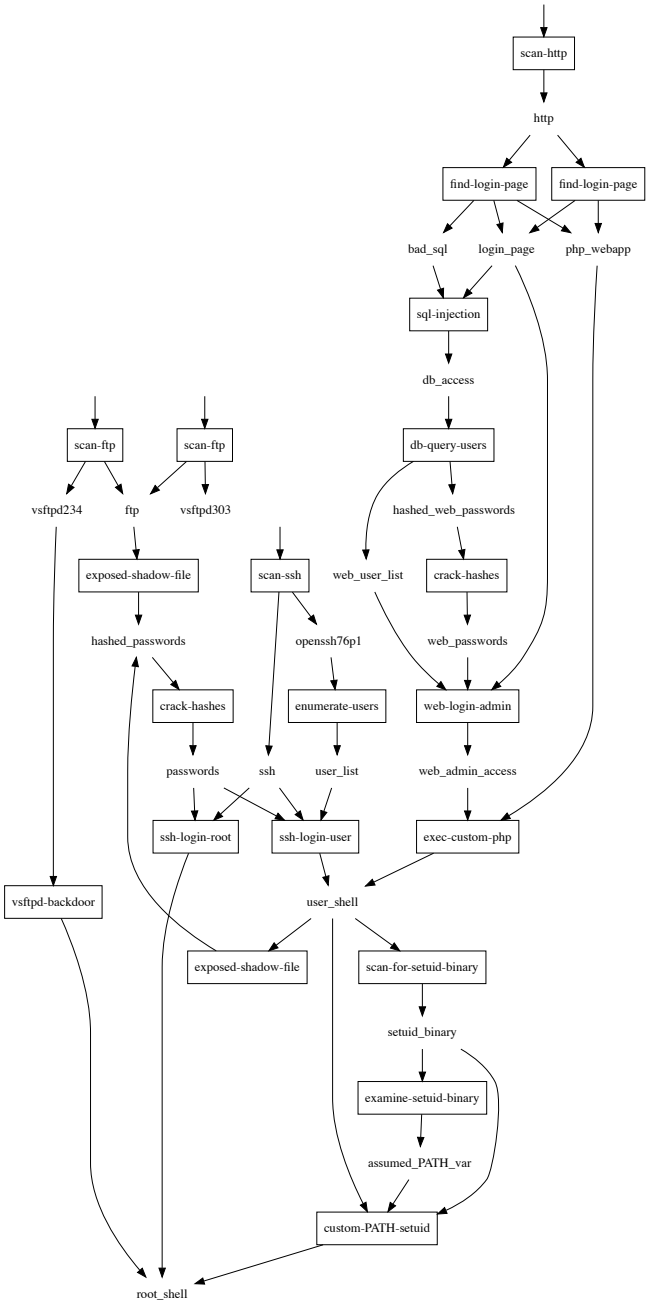

Figure 4: Graphical Depiction of Vulnerabilities (Boxed) and States (Unboxed) Connected by Pre- and Post-conditions. This Graph Represents the Vulnerabilities and States Currently Defined in Alpaca, which are Listed in Table 1

At this time, Alpaca does not provide a simple user interface for running the planning engine. Thus, a user must be mildly familiar

[3] https://www.packer.io
[4] https://www.ansible.com

Vulnerability	Pre-conditions	Post-conditions
scan-ssh	[]	[ssh, openssh76p1]
ssh-login-root	[ssh, passwords]	[root_shell]
enumerate-users	[openssh76p1]	[user_list]
ssh-login-user	[ssh, user_list, passwords]	[user_shell]
scan-ftp	[]	[ftp, vsftpd234]
scan-ftp	[]	[ftp, vsftpd303]
vsftpd-backdoor	[vsftpd234]	[root_shell]
scan-http	[]	[http]
find-login-page	[http]	[php_webapp, login_page]
find-login-page	[http]	[php_webapp, login_page, bad_sql]
web-login-admin	[login_page, web_user_list, web_passwords]	[web_admin_access]
sql-injection	[login_page, bad_sql]	[db_access]
exec-custom-php	[php_webapp, web_admin_access]	[user_shell]
db-query-users	[db_access]	[web_user_list, hashed_web_passwords]
exposed-shadow-file	[user_shell]	[hashed_passwords]
exposed-shadow-file	[ftp]	[hashed_passwords]
crack-hashes	[hashed_passwords]	[passwords]
crack-hashes	[web_hashed_passwords]	[web_passwords]
scan-for-setuid-binary	[user_shell]	[setuid_binary]
examine-setuid-binary	[setuid_binary]	[assumed_PATH_var]
custom-PATH-setuid	[user_shell, setuid_binary, assumed_PATH_var]	[root_shell]

Table 1: List of the Vulnerabilities Currently Represented in Alpaca's Vulnerability Database. Vulnerability Configurations are Omitted for Space. Some Vulnerabilities are Repeated to Provide Different Outcomes; their Configurations Differ to Support these Different Outcomes, and are Defined in Such a Way that Both Outcomes are not Simultaneously Possible. A Graphical Depiction is Shown in Figure 4

with Prolog programming. However, Alpaca is designed to allow arbitrary constraints on the generated paths without modifying the planning engine itself.

4.3 Configuration Merging

The planning engine guarantees that all configurations in a path are compatible. It does this by successively merging configurations as it adds each vulnerability to the growing path. Thus, once a complete path is found, we have a configuration that represents every vulnerability in the path. We call this a path-configuration.

Two configurations may be merged according to the following logic:

- Distinct keys, and their corresponding values, are simply added to the merged configuration.
- For keys that are common between the two configurations, their values (which are also key-value pairs) are merged:
  - Distinct keys, and their corresponding values, are added to the merged set.
  - For shared keys, their values take the form `(X1, List1)` and `(X2, List2)`. We already know these values are compatible from the logic explained in Section 4.1. If `X1=X2`, the new value is `(X1, List1∪List2)`. If `X1=exists` and `X2=only`, the new value is `(only, List1∪List2)`.

4.4 Lattice Generation

The planning engine finds all paths that connect the initial state to the goal state. All vulnerabilities in a single path have compatible configurations, resulting in a merged path-configuration. However, it is not necessarily the case that all paths have compatible path-configurations. In order to generate a lattice, and subsequently generate a cyber range, we must find all subsets of paths that are compatible. Each compatible subset defines a lattice, and the path-configurations of the subset may be merged to form a lattice-configuration. The lattice generation procedure recursively groups paths that have compatible configurations until all paths have been assigned to one or more lattices.

4.5 Cyber Range Instantiation

Given a lattice, cyber range instantiation involves converting the lattice-configuration (which is a merge of all the path-configurations for paths that make up the lattice, each of which is a merge of the vulnerability configurations that make up the path) into Packer and Ansible scripts. Once these scripts exist, creating the cyber range is as simple as running `packer build`, which will create the virtual machine, run Ansible to provision the machine, and then export the machine as a virtual machine appliance.

A lattice configuration is converted to Ansible scripts in the following way. Suppose we have a configuration `[key1-[subkey1-(_, List1)]]`, where `_` indicates we do not care about the distinction between `exists` and `only` at this time. Each key, e.g. `key1`, is a role in Ansible terminology. Each subkey, e.g., `subkey1`, is a variable for the corresponding role, and its values range across the values in its corresponding list, e.g., `List1`. For example, the configuration `[users-[logins-(exists, ["jane", "john"]), passwords-(exists, ["foo", "bar"])]]` turns into the Ansible script:

```
users:
  logins:
    - jane
```

```
    - john
  passwords:
    - foo
    - bar
```

This simple representation of the lattice configuration combines with pre-defined Ansible scripts that, for example, install and configure local users according to the `users` role and variables. Below is an example portion of such a pre-defined Ansible script. It combines the `users.logins` and `users.passwords` variables defined above to generate users with their respective passwords.

```
- name: Create local users
  become: true
  when: >
    users.logins is defined and
    users.passwords is defined
  user:
    name: "{{ item.0 }}"
    password: "{{ item.1 }}"
    state: present
    shell: /bin/bash
  with_together:
    - "{{ users.logins }}"
    - "{{ users.passwords }}"
```

In another example, consider a vulnerability for exploiting a login page with a SQL-injection. In this case, we need to install a particular pre-built web application with a SQL-injection vulnerability (all packaged as a .tar.gz file), as well as the corresponding database tables (as a SQL export). The vulnerability's configuration is as follows:

```
[php-[deployments-(exists, ["loginpage1-badsql"])],
 mysql-[db-(exists, ["logindb1"]),
        root-(only, [generatePassword])]]
```

The generated Ansible portion is shown below. Note, again, that this translation is simple, the configuration is converted into Ansible variables without any preprocessing apart from generating a password.

```
php:
  deployments:
    - loginpage1-badsql

mysql:
  db:
    - logindb1
  root:
    - ah3Xylc
```

The interesting logic is found in the Ansible role scripts, which are predefined for each kind of configuration that is used in the vulnerability database. Adding a new type of vulnerability would require creating a corresponding Ansible role in order to properly install the vulnerability in the cyber range. Some of the Ansible roles for the SQL-injection vulnerability are shown below.

```
- name: Unzip PHP deployments
  unarchive:
    src: "/tmp/{{item}}.tar.gz"
    dest: /
    remote_src: yes
  with_items: "{{php.deployments}}"

- name: Import MySQL databases
  mysql_db:
    name: "{{item}}"
    state: import
    target: "/tmp/{{item}}.sql.gz"
  with_items: "{{mysql.db}}"
```

Alpaca creates one set of Packer and Ansible scripts per lattice. Since multiple lattices may be generated, we organize the scripts for each lattice into a folder hierarchy:

```
ranges/
  [range uuid]/
    range_metadata.json
    [lattice uuid]/
      ansible/
      packer/
      lattice.png
      playbook.yml
```

A single range is created from executing Alpaca. The range is given a universally unique ID (uuid). A JSON file is created containing information about the range, including its starting condition, goal condition, plus all lattice configurations, complexity scores, and vulnerability paths. Then each lattice is assigned a uuid and its scripts are stored in a subfolder with that id. We also generate a visual representation of the lattice and save it to a PNG file. Finally, the requisite Packer and Ansible scripts are copied into the lattice's subfolder. After these folders and files are created, the user may run Packer to generate the virtual appliance for one or more lattices.

5 USE CASES

Alpaca supports several use cases. These use cases cover scenarios ranging across capture-the-flag contests, professional training, and university cybersecurity courses.

Use case 1: A user defines an empty initial state and a specific goal state, e.g., `[root_shell]`. Alpaca produces all lattices that reach the goal state. The user then selects one lattice and Alpaca generates a cyber range that enables that lattice. This cyber range is provided to contestants or students.

Use case 2: A user defines an initial state, e.g., `[php_webapp, login_page]` and a goal state and restricts the generated paths to not include any vulnerabilities that rely on an FTP server. One lattice is selected and the cyber range is generated.

Use case 3: A user defines initial and goal states and generates all possible lattices that have a specified range of complexity. Then he or she generates a cyber range for each lattice, resulting in multiple distinct cyber ranges. Each cyber range is provided to a student or group. If Alpaca's vulnerability database is sufficiently large, it is conceivable that a sufficient number of distinct cyber ranges may be generated for the same initial and goal states to allow each student or group in a course to have a different cyber range.

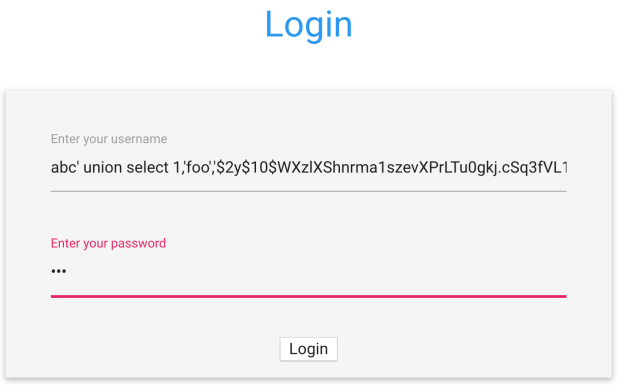

Figure 5: An Example of a Functioning SQL-injection Exploit Enabled by the Generated Cyber Range

6 EVALUATION

Alpaca is a functional proof of concept for building dynamic cyber ranges according to specified constraints. Our evaluation of Alpaca is limited to performance characteristics and a verification that it builds cyber ranges with the expected vulnerabilities.

With respect to performance, given our relatively small vulnerability database, we find that lattices (i.e., paths and configurations) are generated very quickly. Running on a late-2013 MacBook Pro and starting from an initial empty state, lattices are generated in less than one second for various goal states. Likewise, generating Packer and Ansible scripts for a lattice is also nearly instantaneous. Running Packer and Ansible takes considerably more time, on average 25 minutes in our experiments for a full execution starting from an Ubuntu .iso file and ending in a configured virtual appliance. In future work, we plan to support starting from an existing partially configured base system, when desired, which should reduce the cyber range generation time to 4-5 minutes. These times are typical for creating and provisioning virtual machines. Even so, Alpaca is orders of magnitude faster for cyber range instantiation than building one by hand.

Functional evaluation is achieved by building a cyber range from a particular lattice and then confirming that each vulnerability in the lattice is actually exploitable on the cyber range, in the order specified in the lattice. We have confirmed that the cyber ranges do enable the lattices that generate them, though we omit the full details here due to limited space.

Here is an nmap scan demonstrating that the proper services were configured:

```
PORT    STATE SERVICE VERSION
21/tcp  open  ftp     vsftpd 3.0.3
22/tcp  open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.1
80/tcp  open  http    Apache httpd 2.4.29
```

Figure 5 shows a SQL-injection enabled by the cyber range.

7 FUTURE WORK

We see Alpaca as a starting point for a fruitful research agenda in cybersecurity education and practice. We have several plans for future work on this system.

- Using Alpaca requires executing Prolog commands within the Prolog query interface. We envision a web-based user experience that allows a user to select initial and goal states and constraints followed by clicking a "generate" button.
- We plan to add significantly more vulnerabilities to the database, including many CVEs. We also plan to specify vulnerability weights that indicate their difficulty and enhance the complexity measure.
- Currently, Alpaca only supports generation of a single virtual machine. Thus, our cyber ranges do not allow students to practice with host and network discovery, subnets separated by firewalls, and other realistic network-based environments. We plan to introduce network information in configurations and introduce "host discovery" and "subnet discovery" vulnerabilities, among others.
- We plan to allow Alpaca to build cyber ranges from partially configured virtual machines in order to save generation time, and also support Windows virtual machines.

8 CONCLUSION

This paper detailed the design and implementation of Alpaca, a novel dynamic cyber range generator. Unlike previous efforts, this tool focuses on generating cyber ranges that enable multi-step sequences of exploits, thus modeling realistic environments and attack scenarios. Alpaca accomplishes this by using a vulnerability database that defines a vulnerability's pre- and post-conditions and system configuration, and a planning engine capable of chaining vulnerabilities together by their pre- and post-conditions while ensuring their configurations are compatible. Then, these generated paths are grouped together according to their configurations and arranged in a vulnerability lattice, which represents all possible solutions for solving the scenario. A vulnerability lattice may then be transformed into a live cyber range. This is accomplished by generating Packer and Ansible scripts that configure a virtual machine to include all of the vulnerabilities and system configurations required by the lattice. This virtual machine may then be provided to students, who are then challenged to discover the chain of exploits necessary to solve the scenario.

REFERENCES

[1] R. Beuran, C. Pham, D. Tang, K. Chinen, Y. Tan, and Y. Shinoda. 2017. CyTrONE: An Integrated Cybersecurity Training Framework. In *ICISSP*. 157–166.

[2] J. Burket, P. Chapman, T. Becker, C. Ganas, and D. Brumley. 2015. Automatic Problem Generation for Capture-the-Flag Competitions. *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)* (2015).

[3] C. Pham, D. Tang, K. Chinen, and R. Beuran. 2016. CyRIS: A Cyber Range Instantiation System for Facilitating Security Training. In *Proceedings of the Seventh Symposium on Information and Communication Technology*. ACM, 251–258.

[4] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh. 2008. Technical Guide to Information Security Testing and Assessment. *NIST Special Publication* 800, 115 (2008), 2–25.

[5] Z. Schreuders, T. Shaw, G. Ravichandran, J. Keighley, M. Ordean, et al. 2017. Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events. In *USENIX*. USENIX Association.

[6] E. Trickel, F. Disperati, E. Gustafson, F. Kalantari, M. Mabey, N. Tiwari, Y. Safaei, A. Doupé, and G. Vigna. 2017. Shell We Play A Game? CTF-as-a-service for Security Education. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*.