ChainReactor: Automated Privilege Escalation Chain Discovery via AI Planning

Giulio De Pasquale[1,4], Ilya Grishchenko[2], Riccardo Iesari[3], Gabriel Pizarro[2], Lorenzo Cavallaro[4], Christopher Kruegel[2], and Giovanni Vigna[2]

[1]King's College London
[2]University of California, Santa Barbara
[3]Vrije Universiteit Amsterdam
[4]University College London

Abstract

Current academic vulnerability research predominantly focuses on identifying individual bugs and exploits in programs and systems. However, this goes against the growing trend of modern, advanced attacks that rely on a sequence of steps (i.e., a chain of exploits) to achieve their goals, often incorporating individually benign actions. This paper introduces a novel approach to the automated discovery of such exploitation chains using AI planning. In particular, we aim to discover privilege escalation chains, some of the most critical and pervasive security threats, which involve exploiting vulnerabilities to gain unauthorized access and control over systems. We implement our approach as a tool, ChainReactor, that models the problem as a sequence of actions to achieve privilege escalation from the initial access to a target system. ChainReactor extracts information about available executables, system configurations, and known vulnerabilities on the target and encodes this data into a Planning Domain Definition Language (PDDL) problem. Using a modern planner, ChainReactor can generate chains incorporating vulnerabilities and benign actions. We evaluated ChainReactor on 3 synthetic vulnerable VMs, 504 real-world Amazon EC2 and 177 Digital Ocean instances, demonstrating its capacity to rediscover known privilege escalation exploits and identify new chains previously unreported. Specifically, the evaluation showed that ChainReactor successfully rediscovered the exploit chains in the Capture the Flag (CTF) machines and identified zero-day chains on 16 Amazon EC2 and 4 Digital Ocean VMs.

1 Introduction

Current vulnerability research focuses on identifying individual security bugs in the ever-evolving cybersecurity landscape. Several state-of-the-art techniques are employed in this endeavor. Fuzzing [17, 37], for instance, involves providing inputs to a program to detect various vulnerabilities. It is an automated process that can quickly cover a large code base and discover flaws that might be overlooked during manual testing. Symbolic execution [4] is another vulnerability detection technique that systematically explores feasible paths of a program by treating inputs as symbolic values instead of concrete ones. This method effectively uncovers edge-case vulnerabilities by considering heavily guarded execution paths, thus identifying non-trivial conditions that could lead to a security breach. Static analysis [8], both for source code and binaries, is also widely used in vulnerability detection. This approach examines code without executing it to find problematic coding patterns, insecure coding practices, and other potential sources of vulnerabilities.

While some vulnerabilities, such as those in the Linux kernel, can offer an attacker complete control over a system, others are less severe. The presence of a vulnerability does not necessarily translate into a severe threat, as the implementation of anti-exploitation techniques might counterbalance its potential impact. These techniques, such as Control Flow Integrity (CFI) [6] or Address Space Layout Randomization (ASLR) [66], restrict how an attacker can exploit a discovered bug, thus limiting potential damage. Therefore, in modern exploitation scenarios, a single vulnerability may not be sufficient to achieve the attacker's objectives. Security competitions and research efforts targeting real-world software, like Pwn2Own [30] and Google Project Zero [22], highlight the need to develop multi-step exploits.

These steps form a chain where each link, though not necessarily exploiting a critical vulnerability, contributes to the system's eventual corruption. Consequently, the significance of each vulnerability is assessed not merely by the additional attacker capabilities it provides but by its role within the exploitation chain. This includes the number of additional exploits it enables and its contribution to moving the attacker closer to the final step in the desired chain. Furthermore, within the context of an exploitation chain, some benign actions may be necessary for the attacker to get closer to reaching their exploitation goals.

A simple example of the chaining of two vulnerabilities was demonstrated at the Pwn2Own Vancouver event in 2023 [73]. In this case, the attack used an uninitialized variable bug

(CVE-2023-20870 [47]) and a stack-based buffer overflow (CVE-2023-20869 [46]) in VMware Workstation to escalate from a guest OS and execute code in the underlying hypervisor. The first vulnerability involved the misuse of an uninitialized variable within VMware's virtual Bluetooth USB device, leading to an information leak. The second exploit took advantage of a stack-based overflow in the Service Discovery Protocol (SDP) – a feature of the same virtual Bluetooth USB device. Combined, these vulnerabilities allowed the execution of arbitrary code from the guest OS in the context of the hypervisor. This instance illustrates the concept of exploit chains, where multiple vulnerabilities are exploited in a sequence to achieve an attacker's goal, underscoring the need for our approach to automated exploit chain identification.

Currently, the identification of exploitation chains is mostly a manual process. However, the complexity of modern operating systems and execution environments might hide subtle interactions among their components, making it a daunting task for an expert to discern the most suitable path among the myriad possibilities.

To address this issue, we present a systematic approach to automated exploitation chain identification in this paper. Our approach is a classical AI planning problem, encoding the system's initial state (pre-attack) and the goal state (specifying the attacker's objectives). To extract chains, we automate gathering information about the capabilities available to the attacker on the target system. These capabilities represent the possible actions an attacker can perform on the system, from executing specific commands (e.g., downloading content, reading or writing files, and linking specific libraries) to manipulating system configurations or exploiting vulnerable binaries (e.g., binaries with assigned Common Vulnerabilities and Exposures (CVE) numbers [41]). In addition to identifying the presence of standard system binaries (e.g., `cp`), we employ GTFOBins [54], a database that associates specific system binaries with their potential roles in post-exploitation activities (e.g., downloading content). We automatically identify other aspects of system interactions, such as manipulating incorrect permissions in file configurations and system services, which could further expand the attacker's capabilities.

After obtaining the capabilities, we encode them into the initial state of the planning problem as facts. This is done by defining a specific predicate for each capability type and instantiating it. Then, for our planning problem, we select as the goal state the attacker's ability, starting from an unprivileged shell, to gain complete control of the system (i.e., obtaining a root shell) or obtain a shell as another user with higher privileges. ChainReactor does not focus on any particular kind of privilege escalation chain, as the specific instantiation is decided by the planner, which combines the actions based on the facts provided initially, or derived by applying other actions. For instance, if the system has misconfigured permissions, they might be used in the exploit chain, but the planner is not forced to use them.

In this paper, we make the following contributions:

- We introduce the first automated approach for exploit chain discovery based on AI planning.
- We develop a novel method for automated extraction of system programs and configurations, translating them into a format that AI planners can reason with.
- We evaluate ChainReactor on a benchmark of 3 known VMs with privilege escalation bugs and then proceed to test it on 504 Amazon EC2 and 177 Digital Ocean instances as real-world scenarios, identifying 16 and 4 zero-day chains as plans, respectively.
- Lastly, we demonstrate how we can transform the planner's outputs into operational exploitation sequences, obtaining working exploits for all chains that ChainReactor discovered.

2 Background

2.1 Privilege Escalation

Privilege escalation involves exploiting a bug, design flaw, or configuration oversight in an operating system or software application to gain elevated access to resources that are typically protected from (unavailable to) an unprivileged application or user. This can occur in two forms: horizontal privilege escalation, where a user gains the privileges of another user, and vertical privilege escalation, where a user with lower privileges can obtain higher privileges, usually those of a system administrator, also referred to as `root`.

2.2 Planning

Planning Domain Definition Language (PDDL), an expressive language that describes planning domains and problems has become a standard input for AI planners – solvers that generate sequences of actions to transition from an initial state to a goal state [18].

The strength of these AI planners lies in their ability to efficiently search an ample space of possible states and actions to find an optimal (or, at least, satisfactory) plan. PDDL's structure is split into domain and problem files, ensuring a clear distinction between universal problem space aspects and individual problem instance specifics. In the context of our research, this design facilitates the modeling of Unix systems (e.g., user permissions, file hierarchy) and the encoding of attacker goals as problem files, highlighting the potential of AI planners as tools for security analysis.

The *domain file*, serving as a blueprint of the problem space, describes the types of entities or objects in the domain, categorizing them into different classes. These objects, ranging from tangible entities such as robots or tools to abstract concepts such as tasks or states, are instances of specific types

defined within the domain file. In our domain, the entities include system users, groups, different types of files - including executables and directories - and file system permissions. Complementing the objects in the domain file are predicates, representing properties or relationships between the objects. Predicates, essentially Boolean functions, return true or false, depending on the state of the objects they refer to. For instance, a predicate could be `(user_group ?user ?group)`, which is true if the user `?user` is part of the group `?group`, i.e., there is an instantiation with concrete objects replacing `?user` and `?group` parameters that make the predicate true. The domain file further defines the actions or operators that can be performed within the domain. Each action is described in terms of its parameters, preconditions, and effects. Preconditions, often expressed in terms of predicates, specify the conditions for executing the action. At the same time, the effects describe how the action modifies the truth values of certain predicates. Considering an action whose goal is to write to a file, a precondition could be `(executable_can_write_to_file ?executable ?file)`, which needs to be true for an executable `?executable` to be able to write to a file `?file`.

In contrast, the *problem file* specifies a particular problem instance within the defined domain. It provides a concrete definition of the initial state of the domain, describing which predicates are true and false at the beginning of the planning process. The problem file also defines the objects and goal state, usually expressed as a conjunction of predicates, specifying the conditions that must be met to solve the problem.

Together, the domain and problem files form a comprehensive and flexible framework for defining problems in PDDL. The domain file provides the general structure and rules of the problem space, while the problem file offers the specific details of the problem instance.

3 Motivation & Threat Model

An exploitation chain is necessary when achieving an attacker's goal requires multiple steps. This often occurs when the initial access to a system is limited, necessitating a series of actions to compromise the system. In this process, the attacker leverages a combination of specific vulnerabilities and generic system capabilities. These generic system capabilities - such as file access permissions - are integral building blocks for system interaction. Unlike vulnerabilities, which introduce additional or unexpected capabilities, these system capabilities are consistently present and fundamental to the system's operation. These inherent capabilities can provide an attacker with other avenues for exploitation, making them a crucial component of the exploitation chain where every step is designed to escalate privileges or gain additional access, building on the foothold established by the previous step.

3.1 Motivating Example

We present the scenario outlined in Figure 1 to highlight the necessity of chaining vulnerabilities for effective privilege escalation. This scenario is inspired by findings in our experimental evaluation, which revealed multiple Linux systems with CVE-affected binaries and misconfigurations.

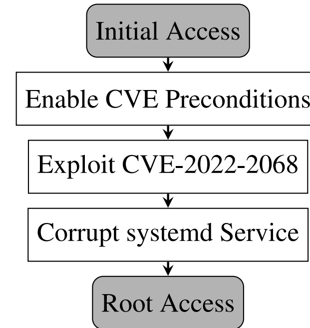

Figure 1: Exploitation steps for the motivating example.

The system in our scenario comprises three users: *Alice*, *openssl*, and *root*. Initially, the attacker controls *Alice*'s account, a basic user account without any special privileges. The goal of the attacker is to obtain root access. In our scenario, the attacker first targets CVE-2022-2068 [44]. This OpenSSL vulnerability arises from the application's failure to properly sanitize shell meta-characters in the file names of certificates being hashed. Under normal circumstances, these files are not directly accessible to non-privileged users, but through a misconfiguration, an attacker might inject malicious content into these file names. After exploiting the OpenSSL vulnerability, the attacker leverages their newly-gained privileges to tamper with a system service file, enabling them to execute code as *root*. The following discussion describes the steps needed to exploit the system in more detail:

1. **Enabling CVE Preconditions** The first finding is a misconfiguration of the directory `/etc/certs/`, inadvertently set to allow write permissions for non-privileged users. Non-privileged users, such as the account controlled by the attacker (*Alice*), should be unable to write or modify files in this directory. This misconfiguration is important because CVE-2022-2068 requires the attacker to feed maliciously crafted file names to OpenSSL. A non-privileged user typically cannot perform this action, due to restricted access to the relevant directories where OpenSSL processes certificate files. By identifying and exploiting this misconfiguration, the attacker can place maliciously crafted files within `/etc/certs/`, creating the necessary precondition for successfully exploiting CVE-2022-2068.

2. **Exploiting CVE-2022-2068** Using the ability to write to the `/etc/certs/` directory, the attacker moves

to exploiting CVE-2022-2068. To this end, they place maliciously-crafted certificate file names in the `/etc/certs/` directory and then trigger the `c_rehash` script, which is executed by the *openssl* user. This script, failing to sanitize shell meta-characters in the file names properly, executes the attacker's code with the privileges of the *openssl* user.

3. **Corrupting a systemd Service** For the third step, the attacker uncovers another misconfiguration: Specifically, the *openssl* user has permission to modify systemd service files. This misconfiguration arises from the fact that systemd service files are incorrectly associated with an administrative group with write permissions to these files, and the *openssl* user is a part of that group. This is not a standard configuration but a possible setup where an admin group is used to manage a variety of system services. The attacker uses this misconfiguration (and the fact that they can run code as the *openssl* user) to inject a malicious modification into the systemd service file. Since the service file runs with *root* privileges when the system or the service is restarted, the modified service file executes code or commands inserted by the attacker. This effectively allows the attacker to escalate their privileges to *root*, granting them complete control over the system.

The exploitation chain highlighted in this scenario leverages a combination of three distinct security issues: the presence of misconfigured directory permissions, the exploitation of a known vulnerability (CVE), and finally, the manipulation of a systemd service for the final escalation to *root*.

Problem Size. The current approach to finding privilege escalation chains requires the manual composition of each step. Some of these steps can be identified by tools such as linPEAS [56] that perform system analysis and report potential vulnerabilities. However, the end-to-end analysis must navigate a vast search space to discover privilege escalation chains automatically. For the exploitation chain we discuss in this section, the analysis must explore millions of possible states, even on basic Linux installations. This complexity is beyond simple brute-force exploration, but different optimized search algorithms can address it. However, any search strategy will require a flexible and extensible mechanism to encode a problem and must support diverse exploitation primitives and attacker goals. We find that PDDL (discussed in Section 2) is a powerful encoding mechanism that enables us to express problems generically. Moreover, it allows us to directly leverage state-of-the-art search algorithms integral to AI planners.

3.2 Threat Model

ChainReactor operates on the premise that the attacker aims to elevate their initial shell access. Methods for securing initial

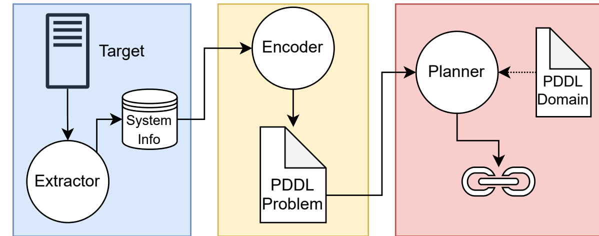

Figure 2: ChainReactor architectural overview.

access vary, ranging from exploiting remote vulnerabilities for service-user access to using stolen credentials for user login.

The attacker's initial capabilities are limited to using the executables present on the system. However, they can retrieve external tools to aid in the exploitation process if they find a way, with no specific constraints on these tools. The attacker does not know about the system architecture or the software running on the machine. However, they can gather this information using the tools available on the system.

Our research primarily focuses on Unix systems, specifically those fully-fledged and equipped with init systems. This current focus is not a fundamental limitation of ChainReactor but rather a design choice to obtain more intricate chains, as init systems (such as [19]) facilitate the automatic execution of routine scripts and executables. Due to this emphasis on init systems, our threat model does not encompass containerized environments.

Moreover, we assume that the attacker initially does not have access to any of the other users' passwords, which rules out the straightforward approach of using privileged commands through the system's built-in permission elevation mechanisms, e.g., using SUDO to assume the identity of another user or log in as that user. However, attackers can exploit known vulnerabilities, e.g., CVEs, to incrementally increase their capabilities.

Furthermore, this work aims to discover privilege escalation chains. It is not limited to any particular kind of privilege escalation chain, as its type (and the involved steps) are decided by automated analysis that combines the actions the attacker can perform on the target system. For example, if there are misconfigured permissions within the system, the planner decides whether to leverage them in the exploit chain.

4 Approach

ChainReactor is built from three main components: the *extractor*, the *encoder*, and the *domain*. The extractor identifies potential vulnerabilities and generic capabilities within the target system. The encoder then converts these capabilities and vulnerabilities into facts formatted in the Planning Domain Definition Language (PDDL). As described in Section 2, the PDDL *domain* outlines the possible actions that the system

can perform, and PDDL *problem* defines the initial state of the system and the goal state to be achieved. ChainReactor then uses an AI planner to solve the problem, the sequence of actions that brings the system from the initial state to the goal state. This sequence is a *plan* corresponding to an exploitation chain. A visual representation of this system can be found in Figure 2. The following sections delve into the specifics of each component.

4.1 Extractor

The extractor is run on the target system under the unprivileged account that the attacker controls according to our threat model discussed in Subsection 3.2. Operating as an unprivileged user, the extractor examines the target system. The tasks performed by the extractor can be categorized into two groups: (1) enumeration of generic capabilities and (2) extraction of vulnerabilities.

Generic Capabilities. To obtain a chain, we require not only knowledge about a system's vulnerabilities but also generic information about it. As part of generic capabilities extraction, our system performs the following tasks:

- *Retrieving System Users and Groups*: Identifies all users and groups on the target system, providing an initial understanding of potential users and groups that an attacker could exploit.
- *Retrieving System Executables*: Identifies all executable programs on the system, including both binary ELF executables and scripts (e.g., shell, Python, and Perl scripts).
- *Retrieving Linked System Libraries*: Identifies all libraries automatically loaded by the system executables extracted in the process. This is done to find libraries with incorrect permissions that could be later corrupted.
- *Retrieving Writable Files and Directories*: Scans for files and directories to which the unprivileged user has write access, emphasizing those not owned by this user, as they may facilitate exploitation.
- *Retrieving SUID / SGID Files*: Identifies all executables for which the SetUID or SetGID bit is set. When executed, such programs run with the permissions of the file owner or group and could be exploited for privilege escalation.
- *Retrieving RC files*: Identifies all bash `.rc` files automatically sourced by shells upon login. If writable, these files can execute arbitrary code as other users.
- *Collecting File Ownership and Permissions*: For every file identified on the system, including executables, scripts, and libraries, the extractor collects information regarding file ownership and permissions. This metadata is crucial for understanding potential security vulnerabilities and access control across the system.
- *Retrieving Background Processes*: Extracts which process is executed by which user. If the process has a vulnerability, this information determines the affected user.

Vulnerabilities. Together with generic capabilities, the extractor also checks for known software vulnerabilities, providing the attacker with additional capabilities on the target system. In particular, the vulnerability extraction includes:

- *Retrieving misconfigured systemd services and cronjobs*: This step identifies all `systemd` [19] services and `cronjobs` [38] running on the system. Misconfigured or vulnerable services and jobs could be exploited to manipulate the system's operations, e.g., to execute a malicious script. Misconfigurations are primarily identified by inspecting configuration file permissions.
- *Mapping Executables to Capabilities Using GTFOBins*: In this step, we leverage GTFOBins [54], a curated database that links specific system binaries to potential post-exploitation tasks. By cross-referencing the executables present on the system with the GTFOBins database, we can identify the potential capabilities that each binary could be used for, such as reading or writing files or downloading content.
- *Retrieving Fine-Grained Version Information of Binaries*: This phase first identifies the CVEs associated with the binaries by utilizing the cve-bin-tool [12]. Subsequently, we access fine-grained data regarding the version and patch status of each program installed using Ubuntu's CVE API [75]. This approach allows us to ascertain whether a system's version has been patched in an Ubuntu image. Armed with this precise information, we can efficiently identify vulnerable programs (CVEs) on Linux Ubuntu distributions.

4.2 Encoder

The encoder is responsible for translating the raw data gathered by the extractor into PDDL format, specifically into PDDL *problems* discussed in Subsection 2.2. These and the domain files constitute the input for the AI planner.

The data received by the encoder holds a comprehensive collection of information about the target system, i.e., generic capabilities and vulnerabilities described in Subsection 4.1. Acting as a translator, the encoder converts this collection into corresponding PDDL objects and predicates.

4.2.1 Encoding Generic Capabilities

The encoder parses the generic capabilities provided by the extractor and maps them to one or more facts. These facts

could include the capability of reading or writing a file (e.g., (cap_write_file usr_bin_busybox), Figure 3, Line 4), downloading or uploading a file, executing a shell command, present system executables (e.g., (system_executable usr_bin_busybox), Figure 3, Line 11) and more. Facts reflect properties of the current system state, such as users, groups, files, file permissions, executables periodically called by users, daemon files, and cross-references between executables and libraries.

4.2.2 Encoding CVEs

We adopt a two-step strategy to enhance ChainReactor by integrating information from the Common Vulnerabilities and Exposures (CVE) database.

First, we manually added support for GTFObin(aries) and 11 CVEs to ChainReactor. The effort involved in adding a new vulnerability or CVE is twofold. First, we need to define the effects of exploiting a CVE on the system state and the attacker's capabilities. Then, we need to translate these effects into predicates (discussed in Subsection 2.2) that the planner understands. If the effects of exploiting a CVE introduce new capabilities that are not currently covered by the domain, we need to introduce new predicates. This process is manual and requires an understanding of the CVE and the domain. Still, it allows us to gradually build up a library of CVEs and associated predicates over time.

Let's consider CVE-2022-1271 [43] that we added to ChainReactor as an example. This CVE enables GZIP to write files arbitrarily. The encoder maps this CVE to a predicate, CAP_CVE_write_any_file, which signifies this capability within the PDDL problem. This mapping informs the planner that if this particular CVE is exploited, then GZIP can be used to write any file to which the user executing GZIP has permission. This effectively extends the writing capabilities of the executable, which may not have been possible without the exploitation of the CVE.

While performing a manual study of CVEs commonly found on recent Linux distributions, we noticed that these vulnerabilities fall into a (small) number of categories that provide similar capabilities to attackers when exploited. For example, exploiting a CVE might allow the attacker to execute arbitrary code as the vulnerable application's user. As another example, the attacker might be able to overwrite files for which they do not have permissions.

Therefore, in the second step, we develop an automated system that can label a given CVE based on the capabilities this vulnerability provides to an attacker (assuming they can successfully exploit it). This approach enables us to encode (in PDDL) the effects of exploiting any CVE for which we can determine a label. This significantly streamlines the process of modeling and incorporating CVEs into our tool.

To facilitate this labeling, we have developed a classifier using GPT-4 Turbo [50]. For a given CVE number, our system first recovers the full CVE description from the National Vulnerability Database (NVD) [42]. Subsequently, the description is included in a prompt that is sent to GPT-4 (via OpenAI APIs [49]). This prompt is written such that it asks GPT-4 to assume the role of a security engineer and assign a label to the CVE description. To ensure consistency between different vulnerabilities, we have defined several different labels (categories), and the prompt includes these labels. The prompt also includes information on what to do in case it is not possible to perform any assignment.

```
1  (define (problem micronix-problem-root)
2      (:objects etc_systemd_system_calyptia_service
           - file usr_bin_busybox - executable
           ubuntu_u - user ubuntu_g - group root_u -
           user root_g - group)
3      (:init
4       (cap_write_file usr_bin_busybox)
5       (controlled_user ubuntu_u)
6       (daemon_file
           etc_systemd_system_calyptia_service)
7       (default_file_permission
           etc_systemd_system_calyptia_service
           FS_READ)
8       (executable_systematically_called_by
           opt_calyptia-core_bin_core-start_sh
           root_u)
9       (file_owner
           etc_systemd_system_calyptia_service
           ubuntu_u ubuntu_g)
10      (file_owner usr_bin_busybox root_u root_g)
11      (system_executable usr_bin_busybox)
12      (user_group root_u root_g)
13      (user_group ubuntu_u ubuntu_g)
14      (user_is_admin root_u)
15      ; more facts...
16     )
17     (:goal (controlled_user root_u)))
```

Figure 3: Simplified example of a PDDL problem file generated by the Encoder component of ChainReactor.

4.2.3 Encoding Example

We use a simplified problem description presented in Figure 3 to exemplify the workings of the encoder. The problem file, automatically generated first by running the Extractor and then the Encoder, consists of three parts: objects – constants used in the predicates (Line 2), init – initial state description (Lines 3-15), and goal – goal state description (Line 17).

Problem objects. The objects defined within the domain represent the entities involved in the problem space, such as the files, users, and executables on the system. Their attributes and relationships dictate the possible actions and the state transitions within the problem scenario.

Initial State. In the initial state description, the encoder puts all the identified objects into facts, i.e., each fact is a predicate instantiation as discussed in Subsection 2.2. To accomplish

this, we equip the encoder with an extendable capability-predicate database. Currently, our database consists of 45 predicates, allowing us to describe information about the system as facts, as we exemplify in the following paragraphs.

A fact with predicate `cap_write_file` (Line 4) is introduced if the extractor finds an executable capable of writing files. In our example, `/usr/bin/busybox` was used to instantiate the predicate and also introduced as an object of type `executable` (Line 2). Together, they express that there is a binary in the system capable of writing files. The same executable (`busybox`) is also a system executable, which is expressed by the predicate `system_executable` (Line 11).

Our system also manages executables periodically invoked by a system's service, indicated by the predicate `executable_systematically_called_by`. In our example, the name of the executable is `/opt/calyptia-core/bin/start.sh`, executed with the privileges of the user `root_u` (Line 8).

We also model the file system, generating the corresponding facts for file ownership and permissions, among others. For instance, the executable discussed above (`/usr/bin/busybox`) is owned by the user and group `root_u:root_g` (Line 10). To map the association between a user and a group, the encoder generates the predicate `user_group` (Line 12). In addition to ownership, we also capture the facts about the permissions, e.g., for a daemon configuration file (Line 6); on this specific system, we generate the fact that the file is read-only (Line 7).

According to our threat model introduced in Subsection 3.2, the system under analysis has an (unprivileged) user controlled by the attacker. In our example, this user is called `ubuntu_u`, which is captured by a fact with the predicate `controlled_user` (Line 5). We also extract the fact that the admin of the system is the user `root_u` (Line 14).

Goal State. The goal state for the PDDL problems generated by the encoder is currently predefined. Specifically, the attacker aims to become the root user, represented by the fact `controlled_user root_u` (Line 17). However, in principle, our system supports other goals, such as data exfiltration.

4.3 Domain & Plan

In Subsection 4.2, we discussed how the encoder produces facts instantiating predicates with objects to obtain a PDDL problem description. As discussed in Subsection 2.2, AI planning also requires another component, namely, domain specification. The PDDL domain is manually developed and includes the predicates' definitions. These predicates are defined in the PDDL domain description. Figure 4 contains examples of such predicate definitions, namely, predicates `cap_write_file` (Line 2) and `system_executable` (Line 3). Predicate specification tells us that instantiation for both predicates requires one element of type `executable`, and this is how they are used by the encoder (Figure 3, Line 4 and Line 11, respectively).

Next, the domain encompasses the description of the planner's possible actions to bring the target system from the initial state in the problem description to the goal state. As described in Subsection 2.2, each action contains typed *parameters*, which can be used in both the predicates and the effects, *preconditions* describing the conditions that needed to be satisfied to apply the action, and *effects* describing the outcome of the application of an action. We formulated the actions to correspond with the specific commands or sequences of commands executed during the development of the multi-stage exploit.

We depict example actions in Figure 4. The first action `write_data_to_file` (Lines 6-16) allows the planner to write to a file (specified by the parameter `?f`) as a step in the attack sequence. To execute the action, there must be an executable `?e` in the target system that can write to files (Line 10), and it should be spawned as process `?p` (Line 11) by a user `?u` with the rights allowing them to write to the file `?f` (Line 12). In addition, the file `?f` that we write to must be different from the executable `?e` used to write to a file. If the planner finds in the `init` state or can derive – by replacing the parameters with the objects – the facts in the preconditions, the action can be executed as a part of the plan. Executing this action leads to the introduction of the effects. In particular, the process `?p` that writes to a file will no longer be active (Line 14), and file `?f` can be moved to location `?l` of choice with data `?d`. It is important to note that the planner chooses the location and the data. Hence, if there is a precondition that can be satisfied if file `?f` is present at location `?l` with data `?d`, the planner can use action `write_data_to_file` to obtain it.

Figure 4 also contains another example action, namely, `spawn_injected_shell_from_corrupted_daemon` (Lines 17-23). It has two preconditions: there should be a daemon `?f` present on the target (Line 20), and the contents of this daemon should contain the attacker-introduced shellcode (Line 21). As the extractor establishes the daemon's presence, the first precondition must be part of the initial state specification. But the second precondition can be derived, for instance, by the previous action `write_data_to_file` application. If both preconditions are satisfied, the effect would be for the attacker to control any user `?u` (Line 23). Thus, this action can be the last step in achieving privilege escalation.

Here, we just highlighted a couple of actions. Our domain specification contains 30 more actions, providing the planner with additional steps that it can use to achieve privilege escalation, e.g., manipulating sensitive files or exploiting the CVEs. This comprehensive approach ensures our system can leverage known vulnerabilities while discovering new exploit combinations. The process of establishing the domain requires a one-time effort. Once accomplished, it can be iteratively utilized to tackle a variety of problem instances, with

```
(:predicates
    (CAP_write_file ?e - executable)
    (system_executable ?e - executable)
    ; ... more predicates
)
(:action write_data_to_file
    :parameters (?p - process ?e - executable ?f -
         file ?d - data ?l - local ?u - user ?g -
        group)
    :precondition (and
      (not (= ?e ?f))
      (CAP_write_file ?e)
      (process_executable ?p ?u ?e)
      (user_can_write_file ?u ?g ?f))
    :effect (and
      (not (process_executable ?p ?u ?e))
      (file_present_at_location ?f ?l)
      (file_contents ?f ?d)))
(:action
    spawn_injected_shell_from_corrupted_daemon
    :parameters (?u - user ?f - file)
    :precondition (and
      (daemon_file ?f)
      (file_contents ?f SHELL))
    :effect (and
      (controlled_user ?u)))
```

Figure 4: Examples of PDDL predicates and actions in the domain description used by ChainReactor.

arbitrary attack objectives expressed through the designated predicates. Furthermore, the extensibility of the domain is facilitated, accommodating the inclusion of additional predicates to address previously unsupported capabilities or actions representing novel steps in the attack sequence.

The combined specifications of the domain and problem (detailed in Subsection 4.2) constitute a valid input for a planner. Once a plan is obtained, it guides the construction of the multi-stage attack. Given that the actions are meticulously aligned with specific commands or sequences, it is plausible to establish a one-to-one correspondence between the plan and the actual attack sequence.

5 Implementation

In this section, we discuss the implementation of ChainReactor, providing some implementation details about the extractor and planner components.

Extractor. The extractor serves as the initial data collection point and is responsible for gleaning necessary information about the system stat, it is fully automated. We developed the extractor in PYTHON. It collects required data through SSH or reverse-shell communication with instances and the execution of a series of commands. We considered leveraging tools like LinPEAS [56] in our Extractor component. We ultimately decided to implement our version, as it provided additional configuration and vulnerability data and was tailored to our exact use case. A crucial function of the extractor is the identification of Common Vulnerabilities and Exposures (CVEs) discussed in Subsubsection 4.2.2. After extracting the CVEs from a target, we associate these CVEs with a class, obtaining a domain that links a CVE class with its effects.

Planner. The critical observation is that the Extractor component (as well as existing vulnerability checkers) only provide basic facts about the system. ChainReactor adds the crucial step where we can reason about combinations of these basic facts to achieve generic multi-stage exploit chains. This is not possible with current tools. To implement this combination step, ChainReactor incorporates an off-the-shelf AI planner that resolves the generated PDDL problem. The planner generates a sequence of actions progressing from the initial to the goal state, thereby constructing an exploitation chain. However, it's crucial to note that the planner does not enumerate all possible plans. Instead, it employs a satisficing strategy, determining the conditions that must be met or the information that must be obtained to reach a satisfactory decision. Once a solution is identified, the planner then focuses on optimizing the cost of that specific solution rather than seeking alternative solutions. If no solution is found, the system is informed accordingly, and the scenario is marked as such.

We utilized Powerlifted, a lifted PDDL planner [14] to enable effective reasoning and planning. Lifted planners operate using abstract representations with objects and potential states, dynamically grounding details during the search process as needed. They work directly on the abstract domain model without fully grounding specific instances and scenarios. This approach eliminates the need for an expensive pre-processing phase and prevents excessive memory usage, which is particularly beneficial when dealing with large state spaces.

The multitude of objects in our domain results from our automatic system information extraction. This leads to a vast state space, making the grounding process extremely expensive and causing an exponential blow-up even before the solving starts. Therefore, while grounded planners might offer faster search times under certain conditions, a lifted planner is necessary to scale to our automatically generated problems and manage the complexity of our domain.

6 Evaluation

The evaluation of ChainReactor was conducted in a two-stage process to ensure its effectiveness and applicability. The first stage involved testing synthetic environments, specifically vulnerable Capture-The-Flag (CTF) Virtual Machines (VMs) published for educational purposes. The second stage aimed at a real-world scenario, evaluating the performance of our system on Amazon Web Services EC2 (AWS) and Digital Ocean (DO) instances.

Our experiments were performed on an Ubuntu 22.04 LTS

```
1 1: (derive_user_can_execute_file apache_u apache_g usr_bin_vim )
2 2: (derive_user_can_execute_file apache_u apache_g opt_my_backup_sh )
3 3: (derive_user_can_write_file apache_u apache_g opt_my_backup_sh )
4 4: (spawn_process apache_u apache_g usr_bin_vim process )
5 5: (write_data_to_file process usr_bin_vim opt_my_backup_sh shell local apache_u apache_g )
6 6: (spawn_injected_shell_from executable_systematically_called_by_user apache_u root_u apache_g
       opt_my_backup_sh process )
```

Figure 5: Plan generated for the CTF VM.

server running on an Intel Xeon CPU E5-2690 3.00GHz with 256GB of RAM. Notably, the PDDL solving process is single-threaded and not CPU-intensive but memory-intensive. The significant RAM capacity allowed batches of 8-9 problem instances to be solved simultaneously [71], and the number of concurrent solver processes was tuned to saturate the available memory for maximum efficiency. The evaluation used a 30-minute overall limit for each planning task.

6.1 CTF VMs

In the first stage, we tested ChainReactor on 3 CTF VMs from a curated list designed explicitly for privilege escalation [72]. These VMs, inherently vulnerable, served as a controlled environment to assess ChainReactor's ability to identify and exploit vulnerabilities. We started with a hands-on approach of manually identifying privilege escalation paths by following published walkthroughs and conducting independent investigations; the walkthroughs served as both a guide and a baseline to measure ChainReactor's ability to detect and exploit vulnerabilities.

This process allowed us to understand the various privilege escalation methodologies commonly used by attackers and identify privilege escalation chains in these controlled settings, demonstrating ChainReactor's ability to detect vulnerabilities and formulate effective escalation paths accurately. Misconfigured SetUID binaries and writeable scripts periodically called by other users were among the most common attack vectors identified in these VMs.

The plan in Figure 5 outlines a sequence of actions that result in a privilege escalation on the CTF VM "Escalate My Privileges" [76]. We consider two users in this scenario: the unprivileged `apache` user, which we control, and the `root` user.

The exploitation chain begins with Line 1 and Line 2, where the planner identifies that the `apache` user, a member of the `apache` group, possesses execution rights for `vim` and a script located at `opt/my_backup.sh`. In Line 3, the planner establishes that the `apache` user has write permissions on `opt/my_backup.sh`. Next, in Line 4, the planner launches a `vim` process under the `apache` user. In Line 5, the planner leverages the write permissions of `apache` on `opt/my_backup.sh` to inject shellcode into the script. Finally, in Line 6, the corrupted `opt/my_backup.sh` script is automatically invoked as part of a cronjob, leading to the execution of the injected shellcode.

The security bug in this scenario is that the backup script, `opt/my_backup.sh`, is both writable and executable by the unprivileged `apache` user. Yet, it is scheduled to run under administrative privileges. This vulnerability is what allows the escalation of privileges to occur.

6.2 AWS EC2 and Digital Ocean Instances

In the second stage of the evaluation, ChainReactor analyzed cloud-hosted instances to understand how our tool performs under conditions representative of typical real-world use cases. We tested 504 Amazon Web Service EC2 (AWS) and 177 Digital Ocean (DO) droplet instances of available Linux images.

Setup. Both official and customized marketplace images were included to provide diversity in software configurations. We retrieved image lists from Amazon's and Digital Ocean's APIs (`describe_images` and `doctl compute image list`). We used the entire Digital Ocean list and a random subset of the AWS images for our testing. Importantly, it should be noted that no modifications were made to these instances after spawning. The instance sizes were `t2.micro` on AWS and `s-4vcpu-8gb` on Digital Ocean. Additionally, port 22 was exposed.

We established SSH access using the default unprivileged user for each instance to simulate the attacker's initial foothold. It is crucial to highlight that in this scenario, we assumed the default user does not have `sudo (ALL:ALL)` permissions and that the user and root passwords are unknown.

Exploring the State Space. To automatically find sequences of actions that lead to privilege escalation, ChainReactor needs to examine many possibilities, both in terms of the different objects involved and the states it must consider. This complexity is shown in Figure 6 for the number of objects and in Figure 7 for the number of states. On average, our system encounters 2,008 objects for AWS images (with a maximum of 16,574) and 897 objects for DO images (with a maximum of 8,374). Regarding the states to be explored, the average numbers are 8 million (with a maximum of 141 million) for AWS and 924 thousand (with a maximum of 85 million) for DO. This level of complexity surpasses that of brute-force exploration.

Generating Facts and Solving Problems. As we explain

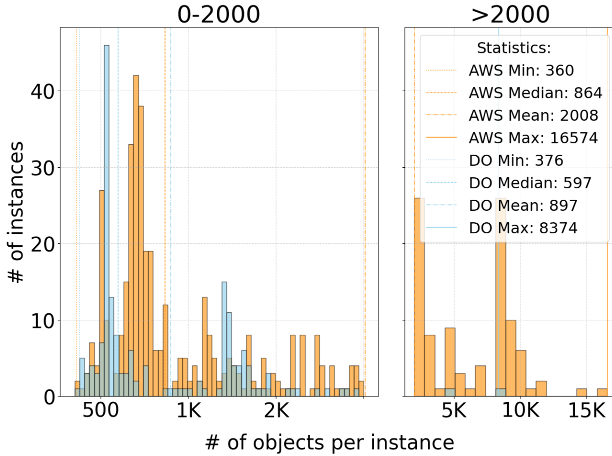

Figure 6: Objects distribution in the generated problems: AWS is orange, and Digital Ocean is blue.

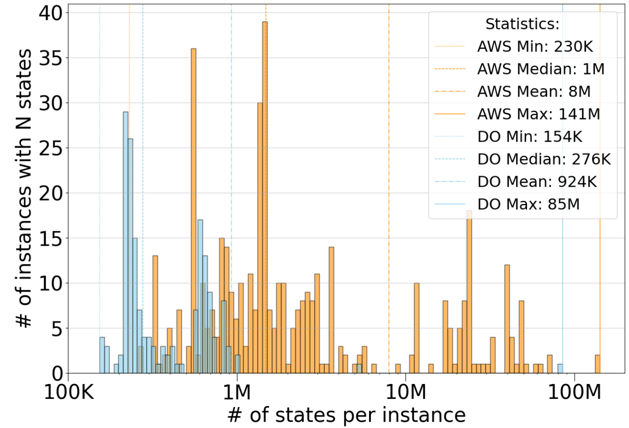

Figure 7: States distribution in the search space: AWS is orange, and Digital Ocean is blue.

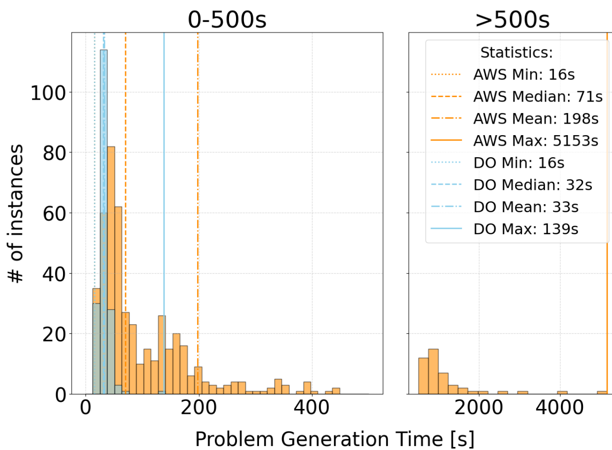

Figure 8: Distribution of time to generate the PDDL problem file: AWS is orange, and Digital Ocean is blue.

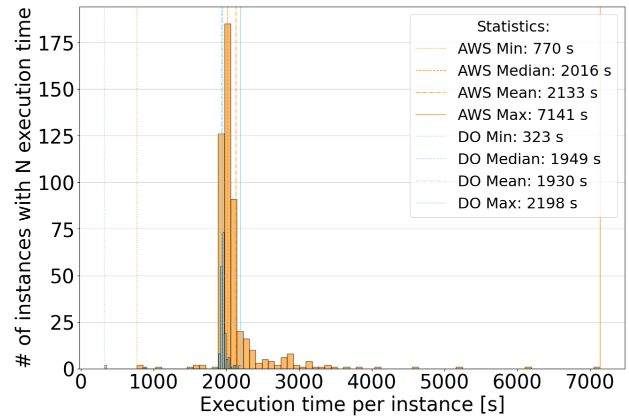

Figure 9: Total analysis time (problem generation + solving) distribution: AWS is orange, and Digital Ocean is blue.

in Section 4, the process with ChainReactor begins with the automated generation of facts about the system being analyzed. This is to set up a problem in the PDDL format. To do this, ChainReactor encompasses the extractor (Subsection 4.1) and the encoder (Subsection 4.2). In Figure 8, we illustrate the time required to create a problem file for each instance. Specifically, creating a problem file with ChainReactor takes, on average, 3 min (max. 86 min) for AWS environments and 33 sec on average (max. 2 min) for DO environments.

Once the PDDL problem is ready, ChainReactor moves on to the planning phase. We show the total time taken, including problem file creation and the planning process, in Figure 9. The average time for processing all EC2 instances, regardless of whether a solution is found, is around 36 min (max. 2h). For DO instances, the average total time for all cases is around 32 min (max. 37 min).

Identified Vulnerabilities. Our evaluation identified several previously unknown privilege escalation chains, specifically 16 targeting AWS and 4 aimed at DO images. Figure 10 displays the duration required to create each chain, arranged from the quickest (5 minutes) to the slowest (half an hour).

We manually reproduced all identified chains. The common thread across these vulnerable instances was the presence of misconfigured permissions. The vulnerabilities we discovered revolved around two main exploitable scenarios. The first involved corrupting a systemd service unit file [19]. It is a configuration file that specifies which executables are to be invoked by systemd, under which user context they will run, and the scheduling of their execution. An attacker with write access to a service unit file can launch a malicious script instead of the intended service. The same can be achieved via the attacker's modifications of a cron-invocable script [38]. The misconfiguration allowed for modifying the daemon file, which, when subsequently executed with root privileges, could

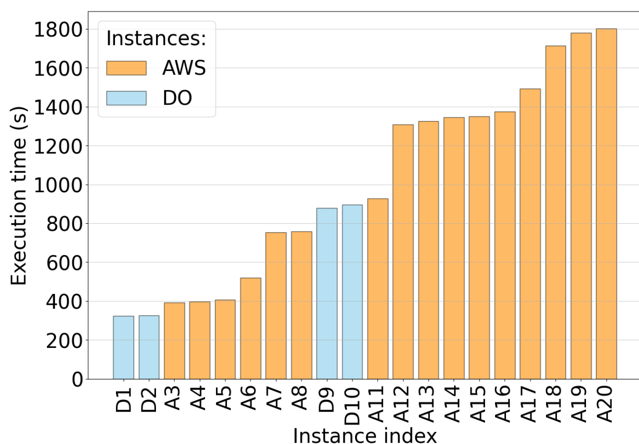

Figure 10: Execution time distribution for exploited instances: AWS is orange, and Digital Ocean is blue.

lead to potentially harmful actions. The second scenario involved the corruption of scripts directly invoked by `root`. Here, too, the misconfiguration of permissions allowed for modifying these scripts, creating a similar risk.

Our two case studies provide a deeper dive into these scenarios, illustrating the risks and potential impacts of these vulnerabilities.

Case study 1. The plan in Figure 11 represents a series of actions that lead to privilege escalation on an EC2 instance. It is important to note that this plan is valid and can be used to exploit the reference instance.

We have two users in this scenario: the unprivileged `ubuntu` user, which we control, and the `root` user.

The chain starts with Line 1, where the planner identifies that the `ubuntu` user, a member of the `netdev` group, has execution rights for `busybox`. This binary combines many common Unix utilities into a single executable. Subsequently, in Line 2, the planner determines that the `ubuntu` user also has write access to the `systemd` service file `calyptia`. In the third step (Line 3), the planner initiates a `busybox` process under `ubuntu` credentials. The fourth step (Line 4) leverages the write access to the `calyptia` service file by injecting shellcode into the `calyptia` service file. Finally, in Line 5, the corrupted `calyptia` service is invoked, leading to the execution of the injected shellcode. Since this `systemd` service runs with `root` privileges, this results in the spawning of a shell as `root`, effectively achieving privilege escalation.

In the presented example, ChainReactor opts for `busybox` to facilitate the exploitation in the outlined steps. However, in other identified chains, we observed the use of different binaries such as `nohup`, `wget`, and `xxd` to achieve similar outcomes.

Case study 2. The plan in Figure 12 represents a series of actions that lead to privilege escalation on another EC2 instance. As in the previous case, this plan is valid and can be used to exploit the reference instance.

We have two users in this scenario: the unprivileged `ec2-user` user, which we control, and the `root` user.

The exploitation chain begins with Line 1 and Line 2, where the planner identifies that the `ec2-user` user, a member of the `docker` group, has execution rights for `vim` and a script located at `opt/aws_dlami/bin/apply_instance_tag_service.sh`. Next, in Line 3, the planner identifies that the `ec2-user` user has write permissions for `opt/aws_dlami/bin/apply_instance_tag_service.sh` script. The planner then launches a `vim` process under the `ec2-user`, denoted by Line 4. In Line 5, the planner leverages the write permissions of `ec2-user` on the script to inject shellcode into this file. Finally, in Line 6, the corrupted `opt/aws_dlami/bin/apply_instance_tag_service.sh` script is assumed to be automatically invoked as part of a routine process, leading to the execution of the injected shellcode. Since the script is executed with `root` privileges, this results in spawning a shell with administrative privileges.

Similarly to the first case study, ChainReactor opted for `vim` while using other binaries to reach the same goal in other chains.

6.3 Expanding ChainReactor with Ubuntu Linux Vulnerability Information

As discussed in Subsection 4.1, our Extractor component has the ability to extract fine-grained vulnerability information for Ubuntu Linux. To demonstrate the effectiveness of incorporating this additional information into ChainReactor, we analyzed all 930 Ubuntu-based Amazon EC2 instances.

We first ran our Extractor component on these instances and identified 264 CVEs. Then, we utilized our GPT-based classifier (Subsubsection 4.2.2), which divided the CVEs into 12 groups. We manually validated that all CVEs within each group indeed introduced the same capabilities, and hence, were labeled correctly. Then, we focused on the largest cluster, containing 120 CVEs, and integrated the corresponding actions into ChainReactor. This process took 6 hours, a significant speedup compared to the initial manual approach, where we needed around the same time to add support for a single CVE.

Upon integrating the 120 new CVEs into ChainReactor, we executed our tool across all Ubuntu-based Amazon EC2 instances. This process unveiled a privilege escalation chain leveraging this action in 20 instances. Specifically, we noted the presence of CVE-2023-45044 [45] in all derived escalation strategies, a vulnerability impacting the Common UNIX Printing System (CUPS) operating as a daemon. Subsequent manual verification confirmed the exploitability of CUPS, thus validating all identified privilege escalation chains in these instances.

```
1 1: (derive_user_can_execute_file ubuntu_u netdev_g usr_bin_busybox)
2 2: (derive_user_can_write_file ubuntu_u ubuntu_g etc_systemd_system_calyptia_service)
3 3: (spawn_process ubuntu_u netdev_g usr_bin_busybox process)
4 4: (write_data_to_file process usr_bin_busybox etc_systemd_system_calyptia_service shell local ubuntu_u
       ubuntu_g)
5 5: (spawn_injected_shell_from_corrupted_daemon_file root_u etc_systemd_system_calyptia_service)
```

Figure 11: Plan generated for the first EC2 instance.

```
1 1: (derive_user_can_execute_file ec2-user_u docker_g usr_bin_vim )
2 2: (derive_user_can_execute_file ec2-user_u docker_g opt_aws_dlami_bin_apply_instance_tag_service_sh )
3 3: (derive_user_can_write_file ec2-user_u ec2-user_g opt_aws_dlami_bin_apply_instance_tag_service_sh )
4 4: (spawn_process ec2-user_u docker_g usr_bin_vim process )
5 5: (write_data_to_file process usr_bin_vim opt_aws_dlami_bin_apply_instance_tag_service_sh shell local
       ec2-user_u ec2-user_g )
6 6: (spawn_injected_shell_from_executable_systematically_called_by_user ec2-user_u root_u docker_g
       opt_aws_dlami_bin_apply_instance_tag_service_sh process )
```

Figure 12: Plan generated for the second EC2 instance.

7 Discussion & Limitations

The following sections delve into a detailed discussion of our research findings. We also acknowledge the constraints of our current approach and identify potential avenues for future enhancements. This balanced examination allows us to better understand our tool's capabilities while setting the stage for further research in automated exploitation chain discovery via AI planning.

7.1 Discussion

A key strength of the ChainReactor system is its ability to model and simulate the behavior of threat actors already present on a target system. This contributes to the field, as many current security measures focus primarily on preventing initial system breaches. Our research addresses a gap in our understanding of advanced persistent threat (APT) actors by shifting the focus to post-breach activities. However, it is essential to clarify that the specific entry points for these threat actors are out of the scope of this paper. Future work could integrate our approach with intrusion detection systems to provide a more comprehensive end-to-end security analysis.

The results from our evaluation suggest that ChainReactor can identify previously unreported chains in addition to rediscovering known privilege escalation exploits. This ability to discover new chains is particularly noteworthy, as it indicates that ChainReactor can adapt to evolving threats and provide valuable insights into potential future attacks.

Responsible Disclosure. Upon identifying exploitation chains, we prioritized the responsible disclosure of these vulnerabilities to the relevant parties, specifically Amazon AWS Security Team [65] and Digital Ocean [48], to mitigate potential security risks. With AWS, our immediate actions led to removing two images from their offerings based on the vulnerabilities we reported.

Throughout this process, we follow the ethical disclosure guidelines, ensuring that no details of live exploits are published or disclosed publicly until they have been addressed adequately by the involved parties. Thus, this work does not contain any information that allows for identifying vulnerable instances.

7.2 Limitations and Future Work

Despite the promising results, there are several limitations to our current approach and areas for future improvement.

First, ChainReactor currently targets privilege escalation but is adaptable to various attacker goals formulated within the PDDL framework. Our suite of test problems extends beyond escalation, covering goals like file exfiltration — utilizing binaries with download/upload functions to transfer files in/out of the compromised system.

Second, the current version of ChainReactor supports a limited set of actions. While the actions supported are sufficient to generate meaningful exploitation chains, expanding the range of actions could lead to the discovery of more complex and subtle exploitation chains.

Third, while ChainReactor can generate exploitation chains, it currently does not produce executable exploit code that can be run directly on the target system. This means that while ChainReactor can identify chains, it cannot automatically validate them by executing them. This is an area for future development, as automatic validation would provide further evidence of the validity of the identified chains.

Fourth, this paper does not cover the initial infiltration phase. ChainReactor's framework could potentially be expanded to include it as a preliminary step.

Last, there are several specific areas of system functionality that ChainReactor does not currently support, including Linux capabilities, network mounts, and path environment variable

injection. These are all potential avenues for exploitation and should be considered in future iterations of ChainReactor.

In conclusion, while ChainReactor represents a significant step forward in automated exploit discovery, work remains. The limitations identified above provide a roadmap for future research and development, with the ultimate goal of creating a fully automated system capable of identifying and validating a wide range of exploitation chains.

8 Related Work

Network Attacks. Various approaches have been proposed to investigate network attacks utilizing Artificial Intelligence (AI). Cohen [11] was among the pioneers in this field, studying attacks as simulations in a cause-effect model. This simulation approach has been further leveraged for evaluating penetration testing practices through planning [3, 27, 68], reinforcement learning [20, 78, 80], Bayesian networks [60], and a blend of Boolean logic and machine learning [28].

Network attacks have also been examined through various modeling techniques. Graph-based models (attack graphs) have been particularly popular, as evidenced by several studies [29, 33, 34, 40, 51, 53, 67]. Other modeling approaches include Petri-net-based models [58, 79], formal logical models [36], and game-theoretic models [1].

AI Planning for Network Attacks. The application of AI planning in the domain of network attack simulation and modeling has proven to be beneficial [3, 27, 39, 68]. Choi et al. [9] applied this concept to smart grid attack response.

"Classical" Single-Step Vulnerability Discovery. This category encompasses a wide variety of techniques. Fuzzing, a method of testing where large amounts of data are inputted into a system, has been examined in several studies [17, 37]. Symbolic execution, a means of analyzing a program to determine what inputs cause each part of it to execute, has also been leveraged [4]. Hybrid fuzzing, which combines fuzzing and symbolic execution, has been explored in depth [7, 55, 69]. Other techniques include bounded model checking, a method for checking properties of programs [5], static analysis, the analysis of computer software performed without executing programs [8], and formal verification, the act of proving or disproving the correctness of programs concerning a certain formal specification or property [25].

Privilege Escalation. Investigations into privilege escalation have spanned across numerous domains, commencing with the early exploration of UNIX's design and security model [23, 63]. Since then, the model has faced a myriad of privilege escalation threats [57, 59], including hardware-targeted attacks [74]. Among the prevalent techniques, the abuse of SetUID permissions has been particularly noteworthy [31, 32], as well as the exploitation of race conditions [16]. In the hardware context, the focus has been on zero-overhead malicious modifications or fabrication attacks [74]. Meanwhile, studies in the browser environment have shed light on privilege escalation attacks via extensions [35]. Trusted Execution Environments (TEEs), despite their promise of enhanced security, have shown vulnerabilities to Horizontal Privilege Escalation (HPE) [70]. The Android platform has also been extensively analyzed for potential privilege escalation threats [2, 15, 61]. Cross-technology performance features have been exploited in the wireless sector for inter-chip privilege escalation [10]. In the cloud environment, potential privilege escalation attack scenarios and misconfiguration vulnerabilities have been identified within Kubernetes and Microsoft's Azure Active Directory [24, 52]. Lastly, investigations into UEFI firmware have utilized protocol-centric static analysis to uncover privilege escalation vulnerabilities in SMM [77].

Grounded and Lifted Planners. To improve the performance of the solving process, planners use preprocessing steps. Among these, a crucial step is grounding. Grounding transforms the problem from the abstract, logic-based PDDL representation to a more concrete, propositional form [13, 21, 26]. Despite their potential efficiency in specific domains, grounded planners are known to be memory-intensive [64]. This is due to the grounding process, which potentially leads to an exponential explosion proportional to the number of predicates, actions, and objects.

Unlike grounded planners, lifted planners operate directly on the abstract, logic-based representation of the problem [14, 62], i.e., they do not perform grounding. While this approach might be less efficient in some scenarios, it provides a significant advantage for larger problems where the resources required for grounding are prohibitive. For our research, dealing with large-scale problems, the immediate search capabilities of lifted planners, despite their potential inefficiency, make them a viable and necessary alternative to grounded planners.

9 Conclusion

This paper presents ChainReactor, an automated system for discovering exploitation chains to achieve privilege escalation on Unix systems. ChainReactor models the problem as an AI planning task, extracting capabilities from the target system, encoding them into a planning problem, and leveraging modern planners to derive exploitation chains.

The evaluation on CTF VMs and real-world Amazon EC2 and Digital Ocean instances demonstrated ChainReactor's ability to rediscover known privilege escalation exploits and identify novel chains. On CTF VMs, it found chains matching or extending published walkthroughs. When evaluated on hundreds of EC2 and Digital Ocean images, it discovered previously unreported exploitable privilege escalation paths.

Overall, ChainReactor offers a practical application of AI planning for security, automating a process to uncover chained exploits.

Acknowledgements

The authors want to express their sincere gratitude to Augusto Blaas Corrêa for his PDDL expertise and support throughout the development of this study.

This material is based on research sponsored by DARPA under agreement number N66001-22-2-4037. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

This material is also supported by the National Science Foundation under grant no. 2229876 and is supported in part by funds provided by the National Science Foundation, by the Department of Homeland Security, and by IBM.

Partial support was also provided through a gift from Cisco.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government, or of NSF or its federal agency and industry partners.

References

[1] Robert K Abercrombie, Bob G Schlicher, and Frederick T Sheldon. Security analysis of selected ami failure scenarios using agent based game theoretic simulation. In *2014 47th Hawaii International Conference on System Sciences*, pages 2015–2024. IEEE, 2014.

[2] Abdulla Aldoseri, David Oswald, and Robert Chiper. A tale of four gates: Privilege escalation and permission bypasses on android through app components. In *European Symposium on Research in Computer Security*, pages 233–251. Springer, 2022.

[3] Adam Amos-Binks, Joshua Clark, Kirk Weston, Michael Winters, and Khaled Harfoush. Efficient attack plan recognition using automated planning. In *2017 IEEE symposium on computers and communications (ISCC)*, pages 1001–1006. IEEE, 2017.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[5] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Handbook of satisfiability*, 185(99):457–481, 2009.

[6] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[7] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2531–2548, 2022.

[8] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004.

[9] Taejun Choi, Ryan KL Ko, Tapan Saha, Joshua Scarsbrook, Abigail MY Koay, Shunyao Wang, Wenlu Zhang, and Connor St Clair. Plan2defend: Ai planning for cybersecurity in smart grids. *2021 IEEE PES Innovative Smart Grid Technologies-Asia (ISGT Asia)*, pages 1–5, 2021.

[10] Jiska Classen, Francesco Gringoli, Michael Hermann, and Matthias Hollick. Attacks on wireless coexistence: Exploiting cross-technology performance features for inter-chip privilege escalation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1229–1245. IEEE, 2022.

[11] Fred Cohen. Simulating cyber attacks, defences, and consequences. *Computers & Security*, 18(6):479–518, 1999.

[12] Intel Corporation. cve-bin-tool. https://github.com/intel/cve-bin-tool, 2023.

[13] Augusto B Corrêa, Markus Hecher, Malte Helmert, Davide Mario Longo, Florian Pommerening, and Stefan Woltran. Grounding planning tasks using tree decompositions and iterated solving. 2023.

[14] Augusto B Corrêa, Florian Pommerening, Malte Helmert, and Guillem Frances. Lifted successor generation using query optimization techniques. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 80–89, 2020.

[15] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Information Security: 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers 13*, pages 346–360. Springer, 2011.

[16] Tanjila Farah, Rashed Shelim, Moniruz Zaman, Md Maruf Hassan, and Delwar Alam. Study of race condition: A privilege escalation vulnerability. In *WMSCI 2017-21st World Multi-Conference Syst. Cybern. Informatics, Proc*, volume 2, pages 100–105, 2017.

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, pages 10–10, 2020.

[18] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[19] freedesktop.org. systemd.unit. https://www.freedesktop.org/software/systemd/man/latest/systemd.unit.html, 2023.

[20] M Ghanem and T Chen. Reinforcement learning for efficient network penetration testing. information 11, 6 (2019).

[21] Daniel Gnad, Alvaro Torralba, Martín Domínguez, Carlos Areces, and Facundo Bustos. Learning how to ground a plan–partial grounding in classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7602–7609, 2019.

[22] Google. Project zero. https://googleprojectzero.blogspot.com/, 2023.

[23] Frederick T Grampp and Robert H Morris. The unix system: Unix operating system security. *AT&T Bell Laboratories Technical Journal*, 63(8):1649–1672, 1984.

[24] Ibrahim Bu Haimed, Marwan Albahar, and Ali Alzubaidi. Exploiting misconfiguration vulnerabilities in microsoft's azure active directory for privilege escalation attacks. *Future Internet*, 15(7):226, 2023.

[25] Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI Global, 2015.

[26] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[27] Jörg Hoffmann. Simulated penetration testing: From" dijkstra" to" turing test++". In *Proceedings of the international conference on automated planning and scheduling*, volume 25, pages 364–372, 2015.

[28] Hannes Holm. Lore a red team emulation tool. *IEEE Transactions on Dependable and Secure Computing*, 20(2):1596–1608, 2022.

[29] Hannes Holm, Khurram Shahzad, Markus Buschle, and Mathias Ekstedt. P^2 CySeMoL: Predictive, probabilistic cyber security modeling language. *IEEE Transactions on Dependable and Secure Computing*, 12(6):626–639, 2014.

[30] Zero Day Initiative. Pwn2own vancouver. https://www.zerodayinitiative.com/blog/2023/3/21/pwn2own-vancouver-schedule-2023, 2023.

[31] Bhushan Jain, Chia-Che Tsai, Jitin John, and Donald E Porter. Practical techniques to obviate setuid-to-root binaries. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[32] Yuseok Jeon, Junghwan Rhee, Chung Hwan Kim, Zhichun Li, Mathias Payer, Byoungyoung Lee, and Zhenyu Wu. Polper: Process-aware restriction of over-privileged setuid calls in legacy applications. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 209–220, 2019.

[33] Pontus Johnson, Robert Lagerström, and Mathias Ekstedt. A meta language for threat modeling and attack simulations. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–8, 2018.

[34] Elmar Kiesling, Christine Strauss, Andreas Ekelhart, Bernhard Grill, and Christian Stummer. Simulation-based optimization of information security controls: An adversary-centric approach. In *2013 Winter Simulations Conference (WSC)*, pages 2054–2065. IEEE, 2013.

[35] Young Min Kim and Byoungyoung Lee. Extending a hand to attackers: Browser privilege escalation attacks via extensions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7055–7071, 2023.

[36] Igor Kotenko. Active vulnerability assessment of computer networks by simulation of complex remote attacks. In *2003 International Conference on Computer Networks and Mobile Computing, 2003. ICCNMC 2003.*, pages 40–47. IEEE, 2003.

[37] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[38] Linux Manual. cron. https://man7.org/linux/man-pages/man8/cron.8.html, 2023.

[39] Doug Miller, Ron Alford, Andy Applebaum, Henry Foster, Caleb Little, and Blake Strom. Automated adversary emulation: A case for planning and acting with unknowns. *MITRE CORP MCLEAN VA MCLEAN*, 2018.

[40] Reuth Mirsky, Ya'ar Shalom, Ahmad Majadly, Kobi Gal, Rami Puzis, and Ariel Felner. New goal recognition algorithms using attack graphs. In *Cyber Security Cryptography and Machine Learning: Third International Symposium, CSCML 2019, Beer-Sheva, Israel, June 27–28, 2019, Proceedings 3*, pages 260–278. Springer, 2019.

[41] MITRE. Cve records. https://cve.mitre.org/, 2023.

[42] NIST-NVD. National vulnerability database. https://nvd.nist.gov, 2023.

[43] NIST NVD. Cve-2022-1271. https://nvd.nist.gov/vuln/detail/CVE-2022-1271, 2022.

[44] NIST NVD. Cve-2022-2068. https://nvd.nist.gov/vuln/detail/CVE-2022-2068, 2022.

[45] NIST NVD. Cve-2023-4504. https://nvd.nist.gov/vuln/detail/CVE-2023-4504, 2022.

[46] NIST NVD. Cve-2022-20869. https://nvd.nist.gov/vuln/detail/CVE-2023-20869, 2023.

[47] NIST NVD. Cve-2023-20870. https://nvd.nist.gov/vuln/detail/CVE-2023-20870, 2023.

[48] Digital Ocean. Hackerone vulnerability disclosure. https://hackerone.com/digitalocean/, 2024.

[49] OpenAI. Openai api. https://openai.com/product.

[50] OpenAI. Gpt-4-turbo. https://help.openai.com/en/articles/8555510-gpt-4-turbo, 2023.

[51] Xinming Ou, Sudhakar Govindavajhala, Andrew W Appel, et al. Mulval: A logic-based network security analyzer. In *USENIX security symposium*, volume 8, pages 113–128. Baltimore, MD, 2005.

[52] Nicholas Pecka, Lotfi Ben Othmane, and Altaz Valani. Privilege escalation attack scenarios on the devops pipeline within a kubernetes environment. In *Proceedings of the International Conference on Software and System Processes and International Conference on Global Software Engineering*, pages 45–49, 2022.

[53] Casey Perkins and George Muller. Using discrete event simulation to model attacker interactions with cyber and physical security systems. *Procedia Computer Science*, 61:221–226, 2015.

[54] Emilio Pinna and Andrea Cardaci. Gtfobins. https://gtfobins.github.io/, 2023.

[55] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[56] Polop. Linpeas - linux privilege escalation awesome script. https://github.com/carlospolop/PEASS-ng/tree/master/linPEAS, 2023.

[57] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[58] Srdjan Pudar, Govindarasu Manimaran, and Chen-Ching Liu. Penet: A practical method and tool for integrated modeling of security attacks and countermeasures. *Computers & Security*, 28(8):754–771, 2009.

[59] Weizhong Qiang, Jiawei Yang, Hai Jin, and Xuanhua Shi. Privguard: Protecting sensitive kernel data from privilege escalation attacks. *IEEE Access*, 6:46584–46594, 2018.

[60] Xinzhou Qin and Wenke Lee. Attack plan recognition and prediction using causal networks. In *20th Annual Computer Security Applications Conference*, pages 370–379. IEEE, 2004.

[61] Mohammed Rangwala, Ping Zhang, Xukai Zou, and Feng Li. A taxonomy of privilege escalation attacks in android applications. *International Journal of Security and Networks*, 9(1):40–55, 2014.

[62] Bernardus Ridder. *Lifted heuristics: towards more scalable planning systems*. PhD thesis, King's College London (University of London), 2014.

[63] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[64] Enrico Scala and Mauro Vallati. Effective grounding for hybrid planning problems represented in pddl+. *The Knowledge Engineering Review*, 36:e9, 2021.

[65] Aamazon Web Services. Vulnerability reporting. https://aws.amazon.com/security/vulnerability-reporting/, 2023.

[66] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, 2004.

[67] Kacper Sowka, Vasile Palade, Hesamaldin Jadidbonab, Paul Wooderson, and Hoang Nguyen. A review on automatic generation of attack trees and its application to automotive cybersecurity. *Artificial Intelligence and Cyber Security in Industry 4.0*, pages 165–193, 2023.

[68] Patrick Speicher, Marcel Steinmetz, Jörg Hoffmann, Michael Backes, and Robert Künnemann. Towards automated network mitigation analysis. In *Proceedings of the 34th ACM/SIGAPP symposium on applied computing*, pages 1971–1978, 2019.

[69] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[70] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. Horizontal privilege escalation in trusted applications. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[71] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[72] Ignite Technologies. Privilege escalation ctf vms. https://github.com/Ignitetechnologies/Privilege-Escalation, 2023.

[73] Nguyen Hoang Thach. Zero day initiative — cve-2023-20869/20870: Exploiting vmware workstation at pwn2own vancouver. https://www.zerodayinitiative.com/blog/2023/5/17/cve-2023-208692087 0-exploiting-vmware-workstation-at-pwn2own-vancouver, 2023.

[74] Nektarios Georgios Tsoutsos and Michail Maniatakos. Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation. *IEEE Transactions on Emerging Topics in Computing*, 2(1):81–93, 2013.

[75] Ubuntu. Ubuntu cves. https://ubuntu.com/security/cves, 2023.

[76] Akanksha Sachin Verma. Escalate my privileges. https://www.vulnhub.com/entry/escalate-my-privileges-1,448/, 2020.

[77] Jiawei Yin, Menghao Li, Wei Wu, Dandan Sun, Jianhua Zhou, Wei Huo, and Jingling Xue. Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1623–1637. IEEE, 2022.

[78] Fabio Massimo Zennaro and László Erdődi. Modelling penetration testing with reinforcement learning using capture-the-flag challenges: Trade-offs between model-free learning and a priori knowledge. *IET Information Security*, 17(3):441–457, 2023.

[79] Gaofeng Zhang, Paolo Falcarin, Elena Gómez-Martínez, Shareeful Islam, Christophe Tartary, Bjorn De Sutter, and Jerome d'Annoville. Attack simulation based software protection assessment method. In *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, pages 1–8. Ieee, 2016.

[80] Shicheng Zhou, Jingju Liu, Dongdong Hou, Xiaofeng Zhong, and Yue Zhang. Autonomous penetration testing based on improved deep q-network. *Applied Sciences*, 11(19):8823, 2021.