A Framework for Automating Security Assessments with Deductive Reasoning

Mauro Andreolini[1,*,†], Andrea Artioli[1,†], Luca Ferretti[1,†], Mirco Marchetti[2,†], Michele Colajanni[3,†] and Claudia Righi[3,†]

[1]*Dipartimento di Scienze Fisiche, Informatiche e Matematiche, Università di Modena e Reggio Emilia, Modena, Italy*
[2]*Dipartimento di Ingegneria Enzo Ferrari, Università di Modena e Reggio Emilia, Modena, Italy*
[3]*Dipartimento di Informatica – Scienza e Ingegneria, Università di Bologna, Italy*

Abstract
Proper testing of hardware and software infrastructure and applications has become mandatory. To this purpose, security researchers and software companies have released a plethora of domain specific tools, libraries and frameworks that assist human operators (penetration testers, red teamers, bug hunters) in finding and exploiting specific vulnerabilities, and orchestrating the activities of a security assessment. Most tools also require minor reconfigurations in order to operate properly with isomorphic systems, characterized by the same exploitation path even in presence of different configurations.

In this paper we present a human-assisted framework that tries to overcome the aforementioned limitations. Our proposal is based on a Prolog-based expert system with facts and deductive rules that allow to infer new facts from existing ones. Rules are bound to actions whose results are fed back into the knowledge base as further facts. In this way, a security assessment is treated like a theorem that has to be proven. We have built an initial prototype and evaluated it in different security assessments of increasing complexity (jeopardy and boot-to-root machines). Our preliminary results show that the proposed approach can address the following challenges; (a) reaching non-standard goals (which would be missed by most tools and frameworks); (b) solving isomorphic systems without the need for reconfiguration; (c) identifying vulnerabilities from chained weaknesses and exposures.

Keywords
Security assessments, Deductive reasoning, Prolog, Isomorphic systems, Automation

1. Introduction

The vast majority of today's services is made available through Internet-based systems that allow for a wide audience through high performance, availability, scalability [1] through a multitude of devices (smartphones, laptops, desktops) [2]. The security of the underlying protocols, system components and interconnects has become of vital importance to managers, designers, programmers, administrators and end users, since a breach might have unforeseen and critical consequences, often leading to business disruption [3]. This increasing attention

ITASEC 2023: The Italian Conference on CyberSecurity, May 03–05, 2023, Bari, Italy
[*]Corresponding author.
[†]These authors contributed equally.

mauro.andreolini@unimore.it (M. Andreolini); andrea.artioli@unimore.it (A. Artioli); luca.ferretti@unimore.it (L. Ferretti); mirco.marchetti@unimore.it (M. Marchetti); michele.colajanni@unibo.it (M. Colajanni); claudia.righi@unibo.it (C. Righi)

to cyber security evidently arises from external threats, which mainly exploit outdated and vulnerable systems, weak security controls and logical bugs. Even worse, inexperienced and insufficiently trained personnel often represents the greatest obstacle to security in the area of cyber crime. Previous studies [4, 5, 6] show that most cyber security incidents result from human errors, misinterpretation of system policies and posture.

Common sense, even before previous research [7, 8, 9], suggests that in various contexts, in order to decrease the risk of security incidents, proper and continuous security assessments need to be conducted on systems. To this purpose, *security assessments* are carried out by *operators* (certified professionals, security researchers, system administrators) with different *goals* in mind: (a) estimate the security risk of an exposed infrastructure (vulnerability assessments) (b) identify and exploit all possible vulnerabilities in a system, showing the consequences of an attack (penetration testing); (c) simulate a skilled adversary that penetrates an infrastructure to achieve a specific mission objective, such as data exfiltration, denial of service, monitoring of specific users (red teaming).
In this complex scenario, security assessments are increasingly difficult to conduct for the following reasons. The attack surface of current systems becomes larger and larger over time. Furthermore, the attack patterns are getting more complex, ranging from structural exploits to lateral movement, cross-domain privilege escalations and exploitation of logical bugs which may not be immediately caught up by popular security tools. To make matters worse, existing software offers little to no support in task planning, definition of non-trivial goals and task orchestration. Finally, most tools also require minor reconfigurations in order to operate properly with *isomorphic systems*, characterized by the same exploitation path even in presence of different configurations.

In this paper we present a human-assisted framework that tries to address the aforementioned limitations. The framework is based on a Prolog expert system managing a knowledge base with facts, deductive rules and associated task templates that should be executed if a rule is found to apply. In this way, a security assessment is treated like a mathematical theorem that has to be proven by applying corollaries and initial conditions. A human operator defines a specific goal and asks if it is achievable with the currently available knowledge. The system applies deductive rules recursively and executes the associated tasks to solve intermediate subgoals. The results of these tasks are translated into new facts and fed back into the knowledge base. Operating in this fashion presents several advantages over current tools:

- it allows to organize existing knowledge about TTPs in a clear, structured, efficient way;
- it allows through deductive rules to define custom assessment goals (e.g., ”Can I exfiltrate all PDF docxuments?”) that go far beyond the classic ones found in CTFs and penetration testing labs (”Am I `root` on this machine?”, ”Can I read `/root/flag.txt?`”);
- it allows to compute a detailed and minimal task plan for a given assessment goal, if sufficient facts and rules are present;
- it allows to solve isomorphic systems by applying the same deductive rules on slightly different facts.

We have built an initial prototype and evaluated it in different security assessments of increasing complexity. Our preliminary experimental results show that the proposed approach can address the following challenges; (a) reaching non-standard goals (which would be missed by most tools and frameworks); (b) solving isomorphic systems without the need for reconfiguration; (c) identifying vulnerabilities from chained weaknesses and exposures.

The remainder of this paper is structured as follows. Section 2 discusses related work and compares our approach with the current state of the art. Section 3 briefly describes the proposed architecture. Section 4 discusses our testbed and some preliminary results. Section 5 concludes the paper with future research directions.

2. Related Work

In this section we discuss and compare our proposal with previous research efforts in the following areas: (a) existing tools, libraries and frameworks serving human operators in security assessments; (b) modeling the activities of a security assessment; (c) defining and enforcing non trivial goals (that go beyond escalating to administrator or capturing a flag); (d) finding complex vulnerabilities that chain multiple weaknesses; (e) solving isomorphic systems.

2.1. Existing tools, libraries and frameworks

A security assessment is carried out through sequence of tasks with the aid of several cyber weapons: *single-purpose tools*, *libraries*, *frameworks* and *sources of information*.

Single-purpose tools. In single-purpose tools (from now on, *tools*) the goal is implicitly hardcoded and, often, domain specific (e.g. Web, binary analysis, networking, privilege escalation). The tool is programmed to automate a specific task and to report on whether a given system component is exposed, vulnerable, exploitable or not. It is the operator's responsibility to orchestrate these tools into a task plan. Popular open source tools include scanners (`Nikto`, `Nmap`), exploitation tools (`SQLmap` [10]), privilege exploitation checkers (`LinPEAS` and `WinPEAS` [11]) and all the proof-of-concept scripts that exploit public CVEs.

Libraries/toolkits Libraries typically provide operators a set of functions for writing their own custom tools with a hardcoded specific goal when the popular ones fail to solve a specific problem. Very often, tools are distributed with the library (in which case the library is also called a `toolkit`). The extra flexibility provided by libraries has a cost in terms of programming skills that an operator must have (this is not always the case). As with tools, the operator is responsible for orchestrating libraries and custom tools into a task plan. Popular libraries are the Python `requests` [12] module, the `pwntools` exploit development package [13], the `Impacket` [14] Python package for low-level interaction with Windows network services.

Frameworks. Frameworks are a coherent collection of binaries, libraries and user interfaces that allows an operator to orchestrate a larger portion of a security assessment. Popular frameworks include BloodHound [15] (to reveal the hidden and often unintended relationships between objects within an Active Directory or Azure environment), textttangr [16] (for binary analysis), Metasploit [17] (to orchestrate reconaissance, vulnerability analysis, exploitation and post-exploitation activities) and its GUI frontend `Armitage` [18]. In contrast to tools and libraries, frameworks offer some support towards attack automation and allow to choose among multiple goals. For example, `angr` uses symbolic execution to achieve the goals specified by a user (e.g., find the code path that crashes an application or reveals the correct password). Here, the goal is explicitely set through programming language primitives and might be obscure to a non expert. `Armitage` provides a *Hail Mary* function that tries all known exploit modules on every single service discovered in a network. Here, attack automation is enforced in a trivial fashion through an implicitly defined goal (find any exploitable vulnerability) which is carried out inefficiently and noisily (it can be easily detected by defenders). `BloodHound` offers a nice graph representation of potential avenues of attack that allow to escalate across Windows domain users, groups and machines. It allows to write arbitrary queries to exhibit custom attack paths.

Sources of information. Sources of informations are documents that explain the internals of the systems under test and the Techniques, Tactics and Procedures (TTP) used to exploit them. They range from unstructured (blogs, papers) to structured ones (checklists, mind maps, attack trees, attack graphs). An operator uses these to define the exact sequence of tasks in a task plan of a security assessment. Building a task plan is usually a manual, tedious, error-prone activity.

2.2. Modeling security assessments

Security assessment activities are often modeled through a complex sequence of intermediate stages, in which privileges are gradually acquired up to being able to reach a specific goal. In such circumstances, it might be difficult to reconstruct the complete attack path and identify the concatenation of techniques and tools. A model is a formal representation aimed at describing the activities of a security professional in terms of techniques used and vulnerabilities exploited in systems and configurations. The purpose of a model is to identify the most probable routes within this sequence. Current literature provides several models of outlined below.

An attack tree [19] represents the attacks to a system and related countermeasures as a tree structure. The root node is the ultimate goal of the attack. Intermediate nodes represent intermediate stages of an attack. Intermediate siblings can be combined in AND or OR mode; AND nodes represent the different steps in achieving a goal, while OR nodes represent different ways to achieve the same goal. Leaf nodes are attacks; they can be labelled to enrich the context of the attack. The notion of attack tree has been extended in literature. Kordy et al. [20] introduce attack-defense trees that also include possible counteractions of a defender. Since interactions between an attacker and a defender are modeled explicitly, this extended formalism allows for a more thorough and accurate security analysis compared to regular attack trees. Zonouz et al. [21] introduce the attack-response tree, basically an attack-defense tree that also includes intrusion detection uncertainties due to false positives and negatives in detecting successful intrusions.

Attack trees can quickly become complex as the number of vertexes and edges increases; in particular, it becomes increasingly expensive to identify all paths from a leaf node to the root node. Keep in mind that in realistic scenarios the number of nodes and interconnections can easily exceed thousands. In these conditions the addition of a single node is sufficient to cause a significant increase in the number of arcs, with a consequent increase in the new possible attack paths. Furthermore, since the root node represents the ultimate goal of the attack, it may be necessary to resort to multiple attack trees to model a complex multi-stage attack.

An attack graph [22] combines information related to network topology, eligible vulnerabilities and exploits available on the assets of an IT infrastructure, by providing a visual representation of the attack paths that an attacker must undertake to achieve specific objectives. An attack graph allows the analyst to highlight the structure of a network and to quickly identify the critical paths most subject to attacks. These activities are essential and preparatory to the subsequent phases of hardening and remediation. An extension of the traditional attack graph is the Bayesian attack graph [23], which adds probabilities to the edges for modeling uncertainty in state transitions between nodes. In particular, edges include the probability of exploitation by an attacker. Therefore, the overall probability of reaching the last state is computed based on the combinations of these probabilities.

Although very useful, these models offer a static view of attacks and mitigations to a system; they do not model the actions a human actually carries out on a live system. On the other hand, our proposal associates template actions to the intermediate stages, thus enabling automated execution of complex assessment paths.

Machine learning techniques such as deep learning and reinforcement learning have been investigated to mimic the behavior of attackers [24, 25, 26, 27]. However, while very interesting, most of these approaches have not yet been tried in realistic environments and model vulnerabilities almost exclusively through CVEs [28].

2.3. Enforcing complex goals

In most security assessment tools and libraries (`SQLmap` [10], `Nikto` [29], `Nmap` [30], `LinPEAS` [11] and `pwntools` [13] to name a few) the goal is implicitly hardcoded and, often, domain specific (Web, binary, network, privilege escalation). Here, setting a specific goal simply does not make any sense; the tool is programmed to automate a specific task and to answer whether a given component is vulnerable or not.

Security assessment frameworks offer improved support for goals to a varying extent. For example, `angr` [16] (a binary analysis framework) uses symbolic execution to achieve the goals specified by a user (e.g., find the code path that crashes an application or reveals the correct password). Here, the goal is explicitely set through programming language primitives and might be obscure to a non expert. `Armitage` [18] (a GUI frontend of the Metasploit [17] penetration testing framework) provides a *Hail Mary* function that tries all known exploit modules on every single service discovered in a network. Here, the goal is implicitly defined, trivial (find any exploitable vulnerability), inefficient and noisy (it can be easily detected by defenders). In contrast, our proposal allows to specify arbitrary goals through precise questions to the knowledge base. Those questions are answered efficiently by the Prolog solver through backtracking.

2.4. Finding complex vulnerabilities

To the best of our knowledge, in the landscape of security assessment software support for chaining existing weaknesses into complex vulnerabilities is limited and, at most, domain specific. In the latter case, chaining of weaknesses is achieved through pluggable modules that are configured manually by an operator. For example, `SQLmap` allows to bypass Web Application Firewall (WAF) filters through so called *tamper scripts* that change the payload accordingly before sending the actual SQL injection. `Pwntools` allows to solve automatically simple buffer overflow vulnerabilities by abusing several weaknesses at once (SETUID binary, missing input validation, missing address layout randomization, address leaks) and building the appropriate payload. `Nmap` can be extended modularly through its Nmap Scripting Engine (NSE) to perform sophisticated, multi-stage enumeration activities. The `Metasploit` framework allows executing "post-exploitation" actions after running an exploit successfully. Typical use cases involve migrating a Meterpreter shell to 64 bit process or adding network routes to internal networks.

In this paper, we use deductive rules to chain dependencies across activities. We think that this is an improvement over the aforementioned strategies in two ways: (a) it allows to chain arbitrary rules, thus modeling also uncommon attack patterns; (b) it is more explainable, since rules are explicitely defined and non ambiguous.

2.5. Solving isomorphic systems

To the best of our knowledge, no existing tool, library or framework can address the challenges behind isomorphic systems efficiently. Most of them require manual human intervention in the form of a slightly different configuration or input parameter. Some tools offer limited, domain specific support. For example, `SQLmap` probes automatically any parameter for potential SQL injections, and can even probe HTTP headers if instructed properly. Unfortunately, a user still needs to input manually the specific URL. The `Pwntools` library is capable of solving specific classes of memory corruption vulnerabilties. The `Angr` framework provides methods for analyzing whole classes of software binaries automatically, but it needs to be explicitly programmed to do so. `Armitage`'s Hail Mary function seems to be the only viable option for solving isomorphic systems, since it tries every possible exploit available over every exposed service. However, as remarked in the previous subsections, it is highly inefficient, noisy and non recursive. Goals are limited to what Metasploit modules can offer.

On the other hand, our approach combines deductive rules and discover of new facts from command outputs to lay down the basis for a more general attack pattern discovery that is independent of the specific system configuration.

3. Proposed architecture

3.1. High level design

Figure 1 shows a high level logical design of the proposed system. The goal here is to point out the subsystems needed to satisfy the claims of the paper. Each block might be implemented in different languages (e.g. Prolog, Python) for convenience or performance reasons.

The *Knowledge base* contains a set of facts and deductive rules. Facts are simple statements about a specific system under test (e.g. "this system has an IP address of 10.10.10.48" or "this system has a SSH service running on TCP port 22"). Deductive rules allow to infer new system facts from existing ones (e.g. "a user can log into the system through SSH if the system is reachable through an IP address, hosts a SSH service and the user has valid user credentials for it").

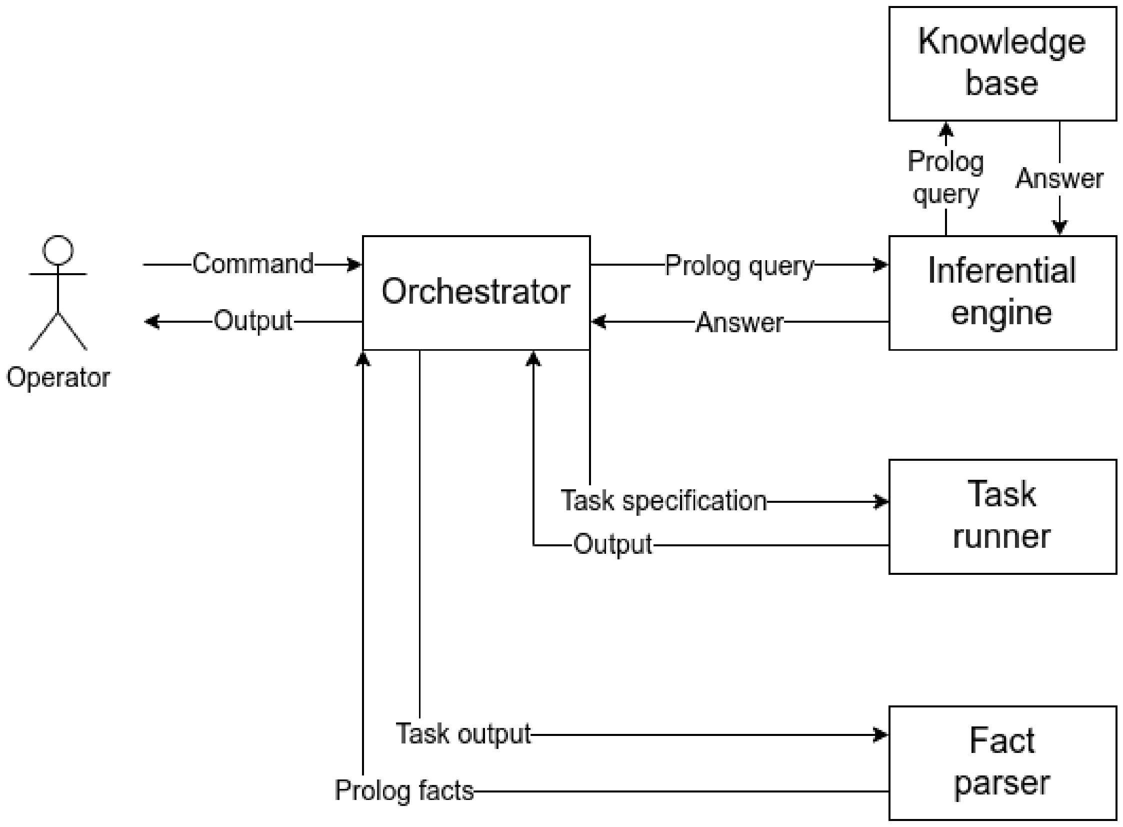

Figure 1: High level logical design

The *Inferential engine* is a Prolog interpreter responsibile for interacting with the knowledge base through proper Prolog queries and collecting the corresponding answers. A user formulates questions about specific system properties by translating them into Prolog queries that check against the existence of specific facts (e.g. ”is this system accessible through SSH?”, ”is there an IP address for which an accessible SSH server is available?”). The inferential engine recursively applies deductive rules related to these facts and de facto splits the query into several, easier subqueries, until these can be trivially answered by already existing facts. The answers to those subqueries are used to construct the final answer, which might be true, false, or a specific text satisfying the requested property (e.g. the IP address of the above mentioned system). We note that this approach applies not only to penetration tests, but to any security assessment that can be logically split into inter-dependent tasks. These include source code security audits, defensive checks and vulnerability research. Furthermore, using rules allows to ask arbitrary goals in form of questions

The proposed approach is static in nature; an operator has to manually input all facts after having verified them through external procedures. We overcome this limitation by binding deductive rules to external commands that are executed whenever the rule is evaluated. The *Task runner* takes in input a task specification (a template command and its parameters, usually fact attributes), assembles a UNIX/Windows command and executes it. Upon command completion it returns a (command output, exit status) tuple. The *Fact parser* receives this tuple, analyzes it and produces in output new Prolog facts that enrich the knowledge base.

The *Orchestrator* connects all the aforementioned components together and defines a set of primitives that may be used to carry on the various procedures involved in a security assessment. Due to reasons of space, we briefly summarize the main interfaces available and discuss the most relevant ones in Section 3.2.

Knowledge base management. CRUD operations (Create, Read, Update, Delete) for rules and facts, load and save a specific Prolog program, load external Prolog modules.

Reasoning. Ask a specific question through a query and obtain an answer, explain a query, obtain query analytics (execution time and steps).

Task execution. Start a task, interrupt a task, obtain status code and output of a task.

Fact parsing. Start a parsing activity, interrupt a parsing activity, obtain facts.

3.2. Implementation details

In this section we discuss the most relevant implementation details of the proposed system. As we will see shortly, almost all modules are implemented in Prolog, the only exception being the fact parsers which are written in Python. The inferential engine used to implement the knowledge base is the popular SWI-Prolog [31]. The knowledge base is implemented as a plain SWI-Prolog program file containing facts and deductive rules. Facts are written as statement tying arguments as follows:

```
HOST_ALIVE('10.10.10.48').
TCP_SERVICE('10.10.10.48',22,ssh,'OpenSSH 6.7p1').
```

These facts state that a host is reacheable at the IPv4 address 10.10.10.48 and it hosts sa specific version of the OpenSSH SSH server. An operator could of course choose another naming convention for facts, as long as it stays consistent across the entire knowledge base. On the other hand, deductive rules require that different facts match simultaneously:

```
SSH_ACCESS(X) :-
    SSH_CREDS(X,Y,Z),
    TCP_SERVICE(X,22,ssh,_).
```

The interpretation of this rule is ”we have SSH access on `X` if the vulnerable system exposes an SSH server on TCP port 22 and we have valid SSH credentials”. The `:-` operator separates the deductive rule from its preconditions. The _ character is a wildcard that matches anything in the corresponding field. In this specific case, the SSH banner is ignored in the rule matching process, since it is irrelevant. When the Prolog engine evaluates this rule, it tries to bind the `X`, `Y` and `Z` variables to parameters defined in existing rules. If a match is found, a new `SSH_ACCESS(X)` fact is generated with `X` bound to the correct IP address.

The previous rule is static in nature; it assumes that if preconditions hold, SSH access is granted, without actually checking them. We can transform this static rule into a dynamic one by attaching an external command to it that will be executed in case the rule is evaluated:

```
SSH_ACCESS(X) :-
    SSH_CREDS(X,Y,Z),
    TCP_SERVICE(X,22,ssh,_),
    process_create(
        path(sshpass), [‘sshpass’, ’-p’, Z, Y, ’@’, X, ‘id’],
        [process(PID)]),
    process_wait(PID, E),
    assertion(E == exited(0))
```

The interpretation of this rule is ”we have SSH access on X if the vulnerable system exposes an SSH server on TCP port 22 and we have valid SSH credentials and those credentials can be actually used to execute a command remotely through SSH”. The `process_create()` library function allows to spawn a new process with the validates SSH credentials by trying to remotely launch the `id` command, automating password input through the `sshpass` command. The `process(PID)` is used to get process ID, which will be used by the `process_wait()` function to synchronously wait for its termination. The exit code is bound to the `E` variable; if the exit code is 0 and every other precondition matches, the rule is valid and a new `SSH_ACCESS(X)` fact is produced with `X` bound to the correct IP address.

The previous rule basically validates a new fact by actually checking it, and expands the knowledge base with it. However, on several occasions an external command might generate a plethora of new facts. A classic example is a network scan with the `nmap` command that produces a set of `TCP_SERVICE` facts (one for each new TCP service discovered). In this preliminary prototype, we pipe such an external command to a filtet written in Python 3. The filter takes in input the output of the external command, analyzes it and produces a series of facts in output. These facts are interpreted by the Prolog environment as new facts. For example, given the following `nmap` output (simplified due to space reasons):

```
    PORT   STATE SERVICE REASON         VERSION
22/tcp open  ssh     syn-ack ttl 63 OpenSSH 6.7p1
53/tcp open  domain  syn-ack ttl 63 dnsmasq 2.76
80/tcp open  http    syn-ack ttl 63 lighttpd 1.4.35
```

the corresponding Python filter would produce the following output:

```
TCP_SERVICE(’10.10.10.48’,22,ssh,’OpenSSH 6.7p1’).
TCP_SERVICE(’10.10.10.48’,53,dns,’dnsmasq 2.76’).
TCP_SERVICE(’10.10.10.48’,80,http,’lighttpd 1.4.35’).
```

Several strategies might be applied in case a task fails. The simplest one does not produce facts; more advanced ones generate facts that signal failures at a specific time, for example to keep track of errors. For the sake of simplicity, in this preliminary version of our prototype each task has its own dedicated fact parser and no rules are produced in case of task failure.

We introduce as an example the `HOST_SCANNED()` rule that applies correctly after a network scan executes successfully:

```
HOST_SCANNED(X) :-
```

```
        process_create(path(nmap_wrap.py),
            [‘nmap_wrap.py’, X, [stdout(pipe(Stream))]]),
        read_stream_to_codes(Stream, Codes),
        close(Stream),
        string_codes(Output,Codes),
        split_string(Output,”\n”,”\n”,Lines),
        maplist(assert_from_string,Lines).
```

The interpretation of this rule is ”host `X` is scanned if `nmap` could be executed correctly through its filter and the output transformed and evaluated to a set of new facts”. Here we execute a `nmap_wrap.py` Python 3 script that runs `nmap`, reads its output and produces the desired facts. The `read_stream_to_codes()` function reads the output from the pipe created with the `stdout(pipe(Stream))` function. The `string_codes()` function converts the byte stream to a string using the system character encoding (usually UTF-8). The `split_string()` function splits the filter output (a sequence of facts, one fact per line) into an array of fact strings. Finally, `maplist()` executes the `assert_from_string()` function to each fact string, de facto inserting it into the knowledge base.

The main interface to the aforementioned functions is SWI-Prolog’s REPL (Read Evaluate Print Loop). Listing all facts and rules in the knowledge base is trivial through the `listing` function:

```
listing().
```

It is also possible to list specific facts or rules by passing them as a parameter to the `listing()` function:

```
listing(HOST_ALIVE).
listing(HOST_SCANNED).
```

Querying for a specific goal is as easy as querying a Prolog fact. For example, to find all alive hosts we can query all known `HOST_ALIVE()` facts:

```
HOSTS_ALIVE(X).
```

The Prolog engine tries to bind all known `HOSTS_ALIVE` parameters to the `X` variable. One possible answer would be:

```
X = ’10.10.10.48’
```

4. Experimental evaluation

In Section 4.1 we define the testbed (hardware, operating system, software) and the systems assessed during the experiments. In Section 4 we compare our prototype with some popular tools and frameworks available to operators. The main goal is to assess strengths and weakness of the prototype.

4.1. Testbed

The testbed architecture running our prototype is based on off-the-shelf hardware and software components. The logic engine is based on SWI-Prolog 9.0.3. All software components are written in Python3, running on a Python3 3.10.9 interpreter. The operating environment is GNU/Linux (specifically, the Arch software distribution). The prototype executes some external commands used during penetration testing activities; those programs have been installed from the corresponding GitHub repositories. In our preliminary evaluation we consider the following systems that encompass a wide range of environments (Web, UNIX, binary).

Web for Pentester. Web for Pentester [32] is a live CD intended to run as a virtual machine. It hosts a series of simple Web-based challenges. Our prototype targets the XSS, SQL injection, Directory Traversal, File Include, Code Injection and Command Injection challenges. These are fairly standard and are solvable with popular tools.

Nebula. Nebula [33] is a live CD intended to run as a virtual machine. It hosts a series of simple UNIX-based privilege escalation challenges based on capture-the-flags. Our prototype targets the Level00 (identification and execution of SETUID binaries) and Level02 (environment variable command injection) challenges. These are not solvable with popular tools and require custom scripts.

Vulnhub Kioptrix. Kioptrix [34, 35] is a series of live CDs intended to run as virtual machines. They are boot-to-root challenges, where one user is supposed to obtain a foothold and escalate privileges to root. The prototype targets Kioptrix1 (which exposes a vulnerable SMB server that yields remote code execution as root) and Kioptrix2 (which exposes a Web server vulnerable to SQL and command injection, and hosts a vulnerable linux kernel). Kioptrix1 is automatically solvable with Armitage, while Kioptrix2 is not. We also built a clone of Kioptrix2 with different configuration parameters (HTTP server port on TCP PORT 8888, vulnerable URLs `/login.php` and `ping.php`, query string parameter `command`) to evaluate tools against structurally similar attack patterns.

HTB Mirai. Mirai [36] is a boot-to-root machine hosted by the HACKTHEBOX gaming platform. It models a Raspberry PI vulnerable to the attacks of the Mirai botnet (default credentials, user in the ”disk” group that can read the whole file system).

4.2. Limitation of existing tools

Web for Pentester challenges are easily solved by common tools such as `SQLmap`, `XSSer` and `commix`. However, these tools are domain-specific and only help the operator in solving subtasks. They do not aid in building an attack plan, nor do they allow to set custom goals. Curiously, `Armitage` cannot solve these challenges due to lack of CVEs on the Web server.

We could not solve any of the Nebula challenges with popular tools and frameworks. We had to resort to custom scripts that use the popular `pwntools` library. While operators could write custom scripts to automate any conceivable assessment, we note that (a) the coding effort is often considerable, (b) the resulting code is often not modular and, thus, not easily reusable in other engagements, (c) goals are hardcoded and attack planning is simply not possible.

Tools and frameworks help only partially in solving boot-to-root machines and need heavy human orchestration. The only exception is Kioptrix1 that hosts a structural CVE on SMB, allowing the operator to escalate privileges to administrator. `Armitage` is capable to solve this box with its Hail Mary attack. Kioptrix2 (and its isomorphic clone) can only be partially exploited in an automated fashion (due to the structural exploit on the Linux kernel). Obtaining foothold on this machine requires orchestrating SQL and command injection tools manually or through custom scripts.

No single tool or framework allows to automate the assessment of Mirai. This machine can be solved through a custom script which would be very complex and with a hardcoded, non reusable attack plan.

4.3. Behavior of the proposed prototype

We prepared an initial knowledge base with all TTPs needed to solve the challenges presented inSection 4.1. The proposed prototype is able to solve them all automatically and can print the attack plan of each assessment. The only input needed is the system's entry point (IP address or URL). Contrary to popular tools and frameworks, the knowledge base can be reused as is in similar security assessments.

Let us discuss some details about the Kioptrix2 boot-to-root machine. Our prototype found the attack plan presented in Figure 2. The graph has been pruned for space reasons and only the relevant paths to privilege escalation have been pointed out. The same attack plan was produced when assessing the isomorphic clone of Kioptrix2, We defined rules to question non

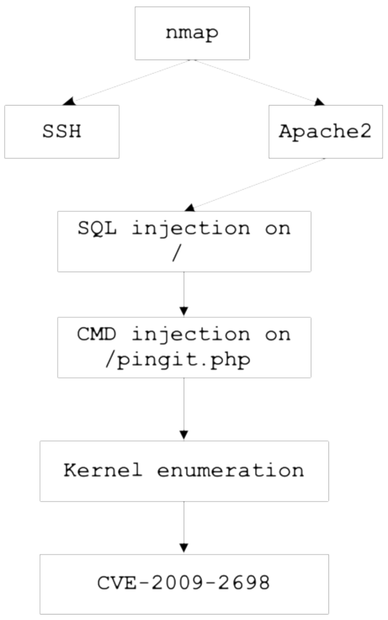

Figure 2: Kioptrix2 attack graph

custom goals, such as finding all PDF files after having escalated privileges to administrator and having found valid SSH credentials. We use a `find_pdf_wrap.py` filter that executes `find /`

`-type f -name *.pdf` remotely through `sshpass`, reads the output and transforms it into several `PDF_PATH(X, PATHNAME)` facts.

```
PDF_FILES(X) :-
    HOST_PWNED(X),
    SSH_CREDS(X, root, Z),
    process_create(path(find_pdf_wrap.py),
        [‘find_pdf_wrap.py’, X, root, Z],
        [process(PID), stdout(pipe(Stream))]),
    read_stream_to_codes(Stream, Codes),
    close(Stream),
    string_codes(Output,Codes),
    split_string(Output,”\n”,”\n”,Lines),
    maplist(assert_from_string,Lines).
```

When queried against, the corresponding output is:

```
PDF_PATH(’10.10.10.48’ ’/root/Documents/Assessment.pdf’).
PDF_PATH(’10.10.10.48’ ’/root/Documents/Invoice.pdf’).
...
```

5. Conclusions and future work

Proper testing of systems is paramount to reduce the risk behind cyber attacks. Security researchers and practitioners have spent most of their efforts in different avenues of work: (a) organizing knowledge through multiple sources of information (both structured and unstructured); (b) modeling attack plans through popular data structures (graphs, trees) and using machine learning techniques; (c) writing domain-specific tools, software libraries, frameworks that aid an operator in enumerating systems, finding and exploiting vulnerabilities. However, we feel that previous efforts lack in several aspects: (a) they are heavily oriented to exploiting public CVEs and basic vulnerabilities, completely neglecting chained ones; (b) they mainly leave to the operator the burden of designing attack plans and specifying non trivial assessment goals; (c) they still exhibit a limited capability of self-discovering future avenues of attack efficiently; (d) they need reconfigurations even in presence of isomorphic systems.

In this paper we present a human-assisted framework that tries to address these challenges. We investigate the possibility of treating a security assessment as a mathematical theorem to be proved. Our proposal is based on a Prolog-based expert system with facts, deductive rules and associated test templates. A human operator asks the system if a specific goal is achievable. Applying deductive reasoning, our prototype tries to solve the goal by building an attack plan. New facts are discovered and fed back into the knowledge base as the assessment proceeds.

Our preliminary experimental results show that the proposed approach can address the following challenges; (a) reaching non-standard goals (which would be missed by most tools and frameworks); (b) solving isomorphic systems without the need for reconfiguration; (c) identifying vulnerabilities from chained weaknesses and exposures.

Our proposal is far from perfect. Maintaining a coherent naming scheme for facts and rules becomes hard their increasing volume. Rule expressiveness is also an issue since Prolog syntax is far from simple. We plan to extend our work in different directions. We will assess the scalability of a single knowledge base and investigate the possibility of implementing a hierarchy of specialized knowledge bases. Furthermore, we will define different evaluation metrics (such as number of attack steps and attack tree depth) to better understand the quality of the attack plan generated by our prototype. We plan to add the concept of priority to single tasks to prefer specific avenues to others. Finally, we will evaluate our prototype to even larger networks involving lateral movements and privilege escalations across several users.

6. Acknowledgments

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

References

[1] H. Lampesberger, Technologies for web and cloud service interaction: a survey, Service Oriented Computing and Applications 10 (2016) 71–110.

[2] A. Jameel, K. Shahzad, A. Zafar, U. Ahmed, S. J. Hussain, A. Sajid, The users experience quality of responsive web design on multiple devices, in: Proceedings of the 2nd International Conference on Future Networks and Distributed Systems, ACM, 2018, p. 69.

[3] K. Huang, M. Siegel, M. Stuart, Systematically understanding the cyber attack business: A survey, ACM Computing Surveys (CSUR) 51 (2018) 70.

[4] B. M. Bowen, R. Devarajan, S. Stolfo, Measuring the human factor of cyber security, in: 2011 IEEE International Conference on Technologies for Homeland Security (HST), IEEE, 2011, pp. 230–235.

[5] M. Evans, Y. He, L. Maglaras, H. Janicke, Heart-is: A novel technique for evaluating human error-related information security incidents, Computers & Security 80 (2019) 74–89.

[6] S. Kraemer, P. Carayon, J. Clem, Human and organizational factors in computer and information security: Pathways to vulnerabilities, Computers & security 28 (2009) 509–520.

[7] W. Knowles, D. Prince, D. Hutchison, J. F. P. Disso, K. Jones, A survey of cyber security management in industrial control systems, International journal of critical infrastructure protection 9 (2015) 52–80.

[8] C. Vellaithurai, A. Srivastava, S. Zonouz, R. Berthier, Cpindex: Cyber-physical vulnerability assessment for power-grid infrastructures, IEEE Transactions on Smart Grid 6 (2014) 566–575.

[9] J. R. Nurse, S. Creese, D. De Roure, Security risk assessment in internet of things systems, IT professional 19 (2017) 20–26.

[10] B. Damele, M. Stampar, SQLmap - Automatic SQL injection and database takeover tool, https://it.wikipedia.org/wiki/Sqlmap, 2006.

[11] C. Polop, PEASS-ng - Privilege Escalation Awesome Scripts SUITE new generation, https://github.com/carlospolop/PEASS-ng/tree/master/winPEAS, 2019.

[12] K. Reitz, Requests: HTTP for Humans, https://requests.readthedocs.io/, 2011.

[13] M. Svensson, Pwntools - CTF toolkit, https://github.com/Gallopsled/pwntools, 2013.

[14] A. Solino, Impacket, https://www.secureauth.com/labs/open-source-tools/impacket/, 2012.

[15] A. Robbins, R. Vazarkar, W. Schroeder, BloodHound: Six Degrees of Domain Admin, https://bloodhound.readthedocs.io/en/latest/, 2019.

[16] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al., Sok:(state of) the art of war: Offensive techniques in binary analysis, in: 2016 IEEE symposium on security and privacy (SP), IEEE, 2016, pp. 138–157.

[17] H. D. Moore, Metasploit - The world's most used penetration testing framework, https://www.metasploit.com/, 2003.

[18] R. Mudge, Armitage - Cyber Attack Management for Metasploit, https://github.com/rsmudge/armitage, 2015.

[19] B. Schneier, Attack trees, Dr. Dobb's journal 24 (1999) 21–29.

[20] B. Kordy, P. Kordy, S. Mauw, P. Schweitzer, Adtool: security analysis with attack–defense trees, in: International conference on quantitative evaluation of systems, Springer, 2013, pp. 173–176.

[21] S. A. Zonouz, H. Khurana, W. H. Sanders, T. M. Yardley, Rre: A game-theoretic intrusion response and recovery engine, IEEE Transactions on Parallel and Distributed Systems 25 (2013) 395–406.

[22] X. Ou, W. F. Boyer, M. A. McQueen, A scalable approach to attack graph generation, in: Proceedings of the 13th ACM conference on Computer and communications security, ACM, 2006, pp. 336–345.

[23] N. Poolsappasit, R. Dewri, I. Ray, Dynamic security risk management using bayesian attack graphs, IEEE Transactions on Dependable and Secure Computing 9 (2011) 61–74.

[24] S. Chaudhary, A. O'Brien, S. Xu, Automated post-breach penetration testing through reinforcement learning, in: 2020 IEEE Conference on Communications and Network Security (CNS), IEEE, 2020, pp. 1–2.

[25] F. M. Zennaro, L. Erdodi, Modeling penetration testing with reinforcement learning using capture-the-flag challenges: trade-offs between model-free learning and a priori knowledge, arXiv preprint arXiv:2005.12632 (2020).

[26] M. C. Ghanem, T. M. Chen, Reinforcement learning for intelligent penetration testing, in: 2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4), IEEE, 2018, pp. 185–192.

[27] R. Maeda, M. Mimura, Automating post-exploitation with deep reinforcement learning, Computers & Security 100 (2021) 102108.

[28] D. R. McKinnel, T. Dargahi, A. Dehghantanha, K.-K. R. Choo, A systematic literature review and meta-analysis on artificial intelligence in penetration testing and vulnerability assessment, Computers & Electrical Engineering 75 (2019) 175–188. URL: https://www.sciencedirect.com/science/article/pii/S0045790618315489. doi:`https://doi.org/10.1016/j.compeleceng.2019.02.022`.

[29] C. Sullo, Nikto, https://cirt.net/Nikto2, 2001.

[30] G. Lyon, Nmap: the Network Mapper, https://nmap.org/, 1997.

[31] J. Wielemaker, An overview of the swi-prolog programming environment, in: Proceedings of the 13th International Workshop on LP Environments, 2003.
[32] PentesterLab, Web for Pentester, https://pentesterlab.com/exercises/web_for_pentester/course, 2012.
[33] A. Griffiths, Nebula, https://exploit.education/nebula/, 2019.
[34] Kioptrix, Kioptrix Level 1, https://www.vulnhub.com/entry/kioptrix-level-1-1,22/, 2015.
[35] Kioptrix, Kioptrix Level 2, https://www.vulnhub.com/entry/kioptrix-level-11-2,23/, 2015.
[36] H. Pylarinos, J. Hooker, A. Zikopoulos, HACKTHEBOX, https://www.hackthebox.com/, 2017.