The Agent Web Model: modeling web hacking for reinforcement learning

László Erdődi¹ · Fabio Massimo Zennaro¹

Published online: 8 June 2021
© The Author(s) 2021

Abstract

Website hacking is a frequent attack type used by malicious actors to obtain confidential information, modify the integrity of web pages or make websites unavailable. The tools used by attackers are becoming more and more automated and sophisticated, and malicious machine learning agents seem to be the next development in this line. In order to provide ethical hackers with similar tools, and to understand the impact and the limitations of artificial agents, we present in this paper a model that formalizes web hacking tasks for reinforcement learning agents. Our model, named *Agent Web Model*, considers web hacking as a capture-the-flag style challenge, and it defines reinforcement learning problems at seven different levels of abstraction. We discuss the complexity of these problems in terms of actions and states an agent has to deal with, and we show that such a model allows to represent most of the relevant web vulnerabilities. Aware that the driver of advances in reinforcement learning is the availability of standardized challenges, we provide an implementation for the first three abstraction layers, in the hope that the community would consider these challenges in order to develop intelligent web hacking agents.

Keywords Agent Web Model · Penetration testing · Capture the flag · Reinforcement learning

1 Introduction

As the complexity of computer systems and networks significantly increased during the last decades, the number of vulnerabilities inside a system has increased in a similar manner. Different types of attackers may try to exploit these varying vulnerabilities for their own benefits. Websites are especially of interest to malicious actors, so attacks against websites nowadays are an everyday event. In order to protect vulnerable systems, one of the best approaches is to emulate real attacks using the same methodology that hackers would use. This practice, named *white hat hacking*, has become a crucial part of critical information technology projects. When taking part into a white hat hacking project aimed at testing the security of a target website, ethical hackers attack the system and report all their findings to the system owner or administrator so that the vulnerabilities can be patched.

Ethical hacking is normally a human job, since the attacker needs a high level of expertise in penetration testing, which involves human capabilities (such as experience, reasoning, or intuition) that are hard to codify.

Although full automation of penetration testing is very challenging, hackers rely on a range of automatic tools [2,12,31] to help them deal with the number and the variety of possible vulnerabilities. In the case of web testing, there are many web security scanners that can help the work of a human tester. These tools can use predefined requests to check the existence of a vulnerability, and quickly generate security reports. However, they have limited capability to carry out complex evaluations, and their findings must normally be reviewed by a human supervisor. Indexes of quality, such as the number of false positives and false negatives, highlight the limited coverage of these tools. New vulnerability detection scripts and general updates may be deployed to improve the performance of web vulnerability scanners, but these are usually one-time solutions lacking automatic improvements. Furthermore, many web scanners are designed only to detect vulnerabilities, but not to exploit them. Specific tools can be used to exploit targeted vulnerabilities with a moderate chance of success [9], and thus advance the understanding of the overall security of the system under study.

✉ László Erdődi
laszloe@ifi.uio.no

Fabio Massimo Zennaro
fabiomz@ifi.uio.no

¹ Department of Informatics, University of Oslo, 0316 Oslo, Norway

Machine learning (ML) techniques aimed at solving problems through learning and inference are now being adopted in many fields, including security [39]. Following their success in challenging tasks like image recognition [22] or natural language processing [41], supervised deep neural network models have been adopted to tackle security-related problems in a static context, such as program vulnerability detection [32] or malicious domain name detection [23]. However, deep neural networks designed to solve static problems exploiting large data sets of examples do not conform to the more complex and dynamic problem of penetration testing. A sub-field of ML that may offer a more relevant paradigm to tackle problems such as web testing, is reinforcement learning. Indeed, reinforcement learning methods allow an agent to learn by itself in a dynamic and complex environment by trial and error and inference. Success on challenging games like Go [37] or Starcraft II [42] suggests that these algorithms may soon find use in the world of penetration testing. Recently, some applications of ML and reinforcement learning in the context of offensive security were developed. On the side of white hat hackers, DARPA organized in 2016 the Cyber Grand Challenge for automated penetration testing [13]. On the side of black hat hackers, malicious bots are being provided with more learning functionalities.

The main motivation behind the current research is to understand and analyze the behavior of ML-based web hacking agents. Since it is inevitable that AI and ML will be applied in offensive security, developing a sound understanding of the main characteristics and limitations of such tools will be helpful to be prepared against such attacks. In addition, such autonomous web hacking agents will be useful for human white hat hackers in carrying out legal penetration testing tasks and replacing the labor-intensive and expensive work of human experts.

However, developing fully autonomous web hacking agents is an extremely complex problem. Replacing a human expert with years of penetration testing experience cannot be done in a single step. This paper aims at fostering this direction of research by studying the way in which the problem of penetration testing may be modeled and decomposed into simpler problems that may be solved by trained reinforcement learning agents. Our modeling effort follows two directions: we first examine the *formalization* of web hacking problems using standard models, and then, we discuss *abstractions* of concrete instances of web hacking problems within our model. We call our generic model the *Agent Web Model*. Aware that a strong and effective driver for the development of new and successful reinforcement learning agents is the availability of standardized challenges and benchmarks, we use our formalization to implement a series of challenges at different level of abstractions and with increasing complexity. We make these challenges available following the standards of the field. Our hope is that these

challenges will promote and advance research in the development of automatic red bots that may help in the tasks of penetration testing.

The Agent Web Model in this paper provides a way to decompose the problem of modeling web hacking into different levels of abstraction with increasing complexity. With this decomposition, we hope to make an important step toward the formalization and the implementation of ML-based web hacking agents from two points of view: first, by providing a potential tentative roadmap of problems with increasing complexity that should be solved in order to develop a web-hacking agent; and, second, by suggesting the Agent Web Model as an interface that may allow researchers in computer security and machine learning to smoothly interact in the definition of problems and in the deployment of RL agents.

This paper is organized as follows. Section 2 presents the main concepts related to web hacking and reinforcement learning. Section 3 discusses how the generic problem of web hacking may be reduced, through a set of formalization steps, to a reinforcement learning problem. Section 4 describes our own model for web hacking problems and describes instances of problems at different level of abstraction. Section 5 explains how real-world hacking problems may be mapped onto the Agent Web Model. Section 6 provides some details on the implementation of challenges based on our formalization. Finally, Sect. 7 discusses some ethical considerations about this work, and Sect. 8 draws conclusions and illustrates possible directions for future work.

2 Background

2.1 Web hacking

The most famous and popular Internet service, the World Wide Web (WWW), has been running for many years [4]. Since its invention in 1989, it had undergone many developments, and nowadays, it is one of the most complex services on the Internet. The HTTP protocol [11] used by these web services has been created for communication within a client-server model. The web client, typically a web browser, sends an HTTP request to a webserver; the webserver, in turn, answers with an HTTP response. An HTTP messages consist of three main parts: the Uniform Resource Locator (URL), referencing the requested object; the HTTP header, containing information on the state of the communication; and the HTTP body, containing the payload of the communication. The request body may contain POST parameters sent by the client, while the answer body usually contains the longest part of the message, that is, the web page content in Hypertext Markup Language (HTTP) format.

Web communication is well defined by the HTTP standard. In time, due to the high number of components

participating in the web communication, the web protocol has become increasingly complex, opening room to different vulnerabilities [44]. A vulnerability can be exploited on the client side, on the server side, or by compromising the communication channel. For instance, attacks against the server side using the HTTP protocol can target the webserver settings, the server side scripts or other resources such as local files or database records. Using the web protocol can thus expose several weak points that can be targeted by malicious actors. The type of the attacks can vary, but they can be categorized according to the information security triplet. Several attacks aim to break *confidentiality* by accessing sensitive or confidential information; others aim at compromising *integrity*, either to cause damage and annoyance or as a preparatory step before carrying out further action; and, finally, attacks may target the *availability* of a service, for instance, overloading a web service with many requests in order to cause a denial of service (DOS).

2.2 Capture the flag

A Capture The Flag challenge (CTF) is a competition designed to offer to ethical hackers a platform to learn about penetration testing and train their skills [25]. CTFs are organized as a set of well-formalized and well-defined hacking challenges. Each challenge has one exploitable vulnerability (or, sometimes, a chain of vulnerabilities) and an unambiguous victory condition in the form of a *flag*, that is, a token that proves whether the challenge was solved or not. Usually, a CTF requires purely logical and technical skills, and they exclude reliance on side channels such as social engineering. Moreover, challenges are normally designed to make the use of brute-forcing or automatic tools unfeasible.

The standard setup of a CTF is the so-called *Jeopardy mode*, in which all players target a single static system. More realistic setups may include the deployment of non-static services with evolving vulnerabilities, or the partition of players in teams, usually a *red team*, tasked with retrieving flags from the target system, and *blue team*, responsible for preventing the attacker from obtaining the flags.

In the case of web challenges, a standard CTF consists of a website hosting objects with different vulnerabilities, and containing flags in the form of special strings. Participants are required simply to collect the flag, and no further exploitative actions are required (such as, setting up a command and control system). Jeopardy-style web CTFs constitute collections of rigorous challenges: the environment in which to operate is well defined, actions can take place only in the digital domain, and objectives and victory conditions are clearly stated. All these properties make CTFs interesting case studies to develop artificial agents for penetration testing.

2.3 Reinforcement learning

Reinforcement learning (RL) is a sub-field of machine learning focused on the training of agents in a given environment [40]. Within such an environment, agents are given the possibility to choose actions from a finite set of available actions. Upon undertaking an action, they can observe the consequences of their actions, both in terms of the effect on the environment, and in terms of a reward signal that specifies how good or desirable is the outcome of that action. The aim of RL is to define algorithms that would allow an agent to develop an action policy leading to as high a reward as possible in time.

The RL problem may be particularly challenging, as the space of actions for the agent may be large, the environment may be stochastic and non-stationary, and the reward signal may be sparse. However, despite these difficulties, RL has been proved successful in tackling a wide range of problems, such as mastering games [26,37] or driving autonomous vehicles [35]. The ability to learn in complex learning environment, such as Starcraft II [42], mirrors the sort of learning that a web hacking agent is expected to perform. RL algorithms may then offer a way to train artificial agent able to carry out meaningful penetration testing.

2.4 Related work

Interest in training artificial red bots able to compete in a CTF challenge has been heightened after DARPA organized a Cyber Grand Challenge Event in 2016 in Las Vegas [13]. In this simplified CTF-like contest, artificial agents were given the possibility to interact with a system exposing a limited number of commands.

However, interest in the problem of modeling and solving hacking or penetration problems predates this event. Different formalizations of CTF-like problems or penetration testing have been suggested in the literature. Standard models relied on formalism from graph theory (e.g., *Markov decision processes* [33]), planning (e.g., *classical planning* [6]), or game theory (e.g., *Stackelberg games* [38]). A wide spectrum of models with varying degrees of uncertainty and varying degree of structure in the action space is presented in [17].

Model-free approaches in which the agent is provided with minimal information about the structure of the problem have been recently considered through the adoption of RL [10,15,29,30]. While these works focus on the application of RL to solve specific challenges, in this paper we analyze the problem of how to define in a versatile and consistent way relevant CTF problems for RL.

A relevant difference between our approach and other published studies is the level of abstraction of the actions.

Many research works model penetration testing by considering high-level actions such as scanning, fingerprinting or exploiting a vulnerability. In [14] and [15], action sets are created from actions like: *probe*, *detect*, *connect*, *scan*, *fingerprint*, *vulnerability assessment*, *exploit*, *privilege escalation*, *pivot*. Similar actions encoding complete vulnerability exploitations (e.g., gaining extra privilege with an existing exploit for a vulnerability identified by the Common Vulnerabilities and Exposures number) constitutes the action space in [8]; a RL algorithm based on a deep Q-network is then used to learn an optimal policy for performing penetration testing. High-level actions such as scans and ready exploits are considered in [34]; standard Q-learning RL algorithm using both tables and neural networks is tested to carry out autonomous penetration testing with the agents being able to find optimal attack paths for a range of different network topologies in their simulated environment. In [28], the set of action includes complex actions such as *scan*, *login*, *enumerate* and *exploit*; multiple algorithms, from fixed-strategy to RL algorithms (Q-learning, extended classifier systems deep Q-networks), are compared. In our approach, we focus on lower-level actions: we consider only simple web requests, and the use of ready exploit as a single action is not an option; for instance, exploiting a SQL injection cannot be carried out in one action. The Agent Web Model aims at building the exploitation process from lower, more basic level than other approaches in the literature.

Notice that, in parallel to this work, some of the problems presented in this paper have already been analyzed and solved with simple RL algorithms in [45] too. In [45], the practical feasibility and the limitation of RL agents were investigated running simplified ad hoc scenarios (e.g., finding and exploiting a service vulnerability with port scanning and an exploitation action). Different from that work, this paper focuses on web exploitation and models the actions at a lower level, as simple web requests; thus, the action of sending an exploit would have to be decomposed into several Agent Web Model actions with different parameters. More importantly, the current work aims at providing a conceptual framework for a wide class of web hacking challenges. Problems tackled in [45] may be reconsidered as particular instances of problems in the Agent Web Model hierarchy; as such, this paper provides a wider and more formalized perspective within which to assess and relate more concrete analyses, like the one offered in [45].

Finally, other researches such as [24] or [3] focus on very specific web cases only. In [24], the action set consists of only POST exploitation actions carried out by PowerShell. In [3], the authors analyzed web application firewalls with a ML-driven search-based approach that combines ML and evolutionary algorithms to automatically detect attacks.

3 Formalization of web hacking

In this section, we explore how the ill-defined problem of web hacking may be *formalized* using different types of standard models (Web hacking $\rightarrow$ CTF $\rightarrow$ game $\rightarrow$ RL problem).

3.1 From web hacking to CTF

As discussed in Sect. 2, real-world web hacking is an extremely complex problem, with vague success conditions and presenting a wide array of possible courses of action, ranging from the exploitation of publicly known vulnerabilities to reliance on non-technical side-channels like social engineering.

CTF challenges represent a way to specify web hacking problems. CTFs offer a clear yet realistic way to define web hacking challenges. There are two important advantages in the modeling of web hacking as CTF: (i) CTF challenges have a well-defined objective, and unambiguous termination conditions (either in terms of flag retrieval or time expiration); and, (ii) CTF challenges define an initial restriction on the actions that can be undertaken by a participant (normally requiring all attempts and attacks to take place in the digital domain).

In this sense, we can understand CTFs as a first step in the formalization of web hacking. However, this formalization is still too loose to be useful for machine learning; most importantly, the space of actions, while being implicitly defined, is still too unconstrained to be useful.

3.2 From CTF to a game

To refine our modeling, we can express CTFs in game-theoretic terms. Web hacking CTFs can be defined as a *game*:

$$\mathcal{G} = \langle \mathcal{P}, \mathcal{A}, \mathbf{u} \rangle,$$

where,

- $\mathcal{P}$ is a set of players,
- $\mathcal{A}$ is a set of actions available to players,
- $\mathbf{u}$ is a vector of utility or payoff functions, such that u_i is the utility function for player i , $1 \leq i \leq |\mathcal{P}|$.

The simplest instance of CTF is a 2-player game with $|\mathcal{P}| = 2$, where one player is the attacker and the second player is the webserver. As long as the web CTF challenge is static, the webserver may be conceived as a player deterministically reacting to the actions of the attacker. As explained in Sect. 2.2, this basic CTF setup may be extended to *adversarial multiplayer games* with $|\mathcal{P}| = N$, where players are partitioned in a red team and a blue team. In the following, we will focus our attention and our discussion on the 2-player

game, although our considerations apply straightforward to the multiplayer case.

For any player, we assume the set of action $\mathcal{A}$ to be finite or countable, so as to allow an artificial agent to select its actions. Notice that this assumption of finiteness or countability is reasonable as long as a CTF takes place in a digital and discrete domain.

The utility function u_i of a player allows for the encoding of the victory condition expressed by a CTF challenge. A stark binary utility function allows to assign a positive utility to the capture of the flag, and a null utility to everything else. More refined utility functions may allow to shape the behavior of a learned agent more subtly.

A game-theoretic formalization can then be seen as a further step in the process of formalization of web hacking problems. The main contribution in this form modeling, contrasted with a generic CTF model, is the definition of an enumerable set $\mathcal{A}$ of possible actions. This provides the foundation for an agent to choose actions and learn its own action policy. Although game theory already provides tools to analyze web hacking as we have modeled it, this formalization is still not ideal as the modeling of a webserver as an active player results over-generic. In the case of interest, in which we have a single attacker targeting a static system, it would be more practical to describe the webserver as a static component of the game.

3.3 From a game to a RL problem

In the case of web hacking with a static system, the game-theoretic modeling over-defines the webserver by describing it as a player. Alternatively, we can model the game as a *RL problem*:

$$\mathcal{R} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle,$$

where

- $\mathcal{S}$ is a set of states the game may be in,
- $\mathcal{A}$ is a set of actions,
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is a state transition function defining how states evolve given an initial state and an action,
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is a reward function defining the reward obtained by an agent after taking an action in a given state.

A RL problem thus defined implicitly assumes a single player. In this model, the webserver is not represented as a second player, but its internal logic is implemented in the state transition function $\mathcal{T}$. The state transition function specifies how the system reacts upon the action of the playing agent, and its dynamics relies on two assumptions. First, we assumed that, in general, the result of

an action $a \in \mathcal{A}$ depends not only on the action itself, but also on the current state $s \in \mathcal{S}$ of the system. This correspond to the assumption of a *stateful* system. This assumption is meaningful, as real web systems may be in different states after interacting with their users. Notice that a stateless system can, in any way, be considered as a limit case of a stateful system with a single unchanging state. Second, we assumed that, in general, the result of an action $a \in \mathcal{A}$, given the current state $s \in \mathcal{S}$, may be *stochastic*. This assumption is meaningful in that real web systems may rely on stochastic functions. Moreover, such an assumption may allow us to model potential network communication fails or attempts by the system to obfuscate its logic. Notice that a deterministic state can, in any way, be considered as a limit case of a stochastic system with a delta distribution function. In sum, we express the logic of the webserver as a probabilistic transition function $\mathcal{T} = P(s'|s, a)$ specifying a probability distribution over future states s' , given the current state s and action a . We will refer to $\mathcal{T}$ as the transition function, the logic of the game, or the dynamics of the environment.

As in the game-theoretic formulation, the set of action $\mathcal{A}$ is a countable set of actions available to the agent.

The reward function $\mathcal{R}$ translates the utility function u from the game-theoretic modeling to the RL formalism.

Finally, the set of states $\mathcal{S}$ allows for the tracking of the state of the game. Notice that although the state of the game is uniquely determined at any point in time, the agent may not be aware of it. This leads to a *partially observable* game, in which the agent has no certain knowledge about the current state of the system, but it has only belief over the possible states. Through its own local state, which encodes its imperfect knowledge, the agent tries to keep track of the actual state of the system. Notice that a completely observable game may be considered as a limit case in which all the beliefs collapse into delta functions.

This final RL formalization captures well enough the problem of web hacking: it is flexible enough to accommodate very different hacking challenges, but, at the same time, is constrained enough that all its component are well defined so that standard RL algorithms may be used to train artificial agents. We will then make the RL formalization the theoretical blueprint of our model for web hacking.

4 The Agent Web Model

In this section, we use the RL formalism defined in Sect. 3 to characterize our own model for web hacking. We then discuss how this generic model may be used to implement actual web hacking problems at different levels of abstraction.

4.1 The Agent Web Model

In order to define a RL problem, it is necessary to define the state transition function of the problem. In our context, this function represents the logic of the target webserver. Different systems, with different types of vulnerabilities, may be represented in different ways. To simplify the modeling of a webserver, we will represent it as a collection of generic *objects*. These objects are taken to represent entities of interest (e.g., files, ports) that can be targeted by the actions $\mathcal{A}$ of an attacker. This simplification allows us to decompose the design of a target system, its logic and its states. Transition functions can be defined in a modular way with respect to specific objects, and the state of the system may be factored in the state of single objects.

The decomposition of a webserver into a collection of objects also allows us to easily define instances of webserver at different *levels of abstraction*. By defining the nature and the number of existing objects, and by defining which actions an agent can take in relation to the defined objects, we can immediately control the complexity of the RL problem at hand.

Moreover, another aim of ours in having a modular system defined in terms of individual objects is the possibility of instantiating new challenges in an automatic, possibly random, way. Such a *generative model* of web-hacking problems would provide the opportunity to easily generate a large number of problems on which to train a RL agent.

We call this flexible, generative model to instantiate different types of web hacking problems, the *Agent Web Model*.

4.2 Levels of abstraction

Concretely, we define 7 different levels of abstraction for web hacking with increasing complexity. Figure 1 offers a visual summary of these levels, together with the essential features of each one. Notice that complexity increases in terms of the actions and the feedback that the agent can receive. Higher level allows for a more detailed modeling by providing the agent with a larger set of actions and/or with actions allowing for multiple parameters. However, increased complexity induces more computationally challenging problems; whenever feasible, we provide an approximate estimate of this computational complexity in terms of the number of actions and the number of states an agent is supposed to deal with. Level1 starts with the model of a very simple website, composed of basic files, abstracting away web parameters and sessions. At higher levels, the agent is expected to interact with more complex objects making up the website; for instance, requests to files can accept multiple input parameters with different web methods, as well as multiple session values.

Fig. 1 Levels of abstraction in the Agent Web Model

A hacking problem at level1 has a trivial solution which could be coded manually in a simple algorithm, but we will show that the computational complexity soon escalates as we move up in the levels. A hacking problem at level7 is close to real-world web hacking, where an attacker can even create its own objects on the target site (e.g., command script) and carry out complex exploitation strategies; this sort of problem is far from a trivial solution.

In the following, we discuss the details of the different layers of the Agent Web Model, including the number of states and actions that have to be handled in different levels. Except when explicitly stated, in all levels of abstractions we will assume that the objects on a webserver are files, and we will take a simple binary reward function $\mathcal{R}$ that returns a unitary reward when the agent accomplishes its task, and zero otherwise.

4.2.1 Level1: link layer

In level1, a website is composed of a set $\mathcal{O} = \{\text{file}_1, \text{file}_2, \dots, \text{file}_N\}$ of objects representing simple static HTML files. We take the first file to represent the *index.html* file inside the webroot. Files are linked to each other by pointers, and one of the files contains the flag. All the files can be accessed by the agent without restrictions; no parameters are required, and the HTTP headers have no meaningful information such as sessions. The actual file content is irrelevant, except for

the case of the flag. Practically, level1 problems can be represented as a directed graph of files (see Fig. 2).

The set of actions comprises only two parametric actions: $\mathcal{A} = \{\text{read}(\text{file}_i), \text{search}(\text{file}_i)\}$. The action $\text{read}(\text{file}_i)$ reads the i^{th} file and returns the list of linked files. The action $\text{search}(\text{file}_i)$ checks the i^{th} file for the presence of the flag. See Table 1 for a summary of the actions, their parameters and their return values. Note that these actions can be performed only on files that the agent has discovered on the remote webserver.

Without training a RL agent, a simple heuristic solution to this problem would be to read the files one by one in order to discover all files, and then search for the flag inside each one. The number of files N that a website hosts has a significant influence on the problem scale. The actual size of the action space $|\mathcal{A}|$ depends on the value of N : an agent can take up to $2N$ different actions, that is, a $\text{read}()$ action and a $\text{search}()$ action for each file. Moreover, an agent is required to keep track of its own knowledge state, that is record what actions has been executed and what result was observed. A basic agent can simply track, for each file, whether action $\text{read}()$ was tried (2^N states) and whether action $\text{search}()$ was tried (2^N states). In total, it will have 2^{2N-1} states; Table 2 shows an estimate of the number of actions and states as a function of the number of files.

4.2.2 Level2: hidden link layer

In level2, we model again the website as a collection of static HTML files. Files are still linked by pointers, but we now distinguish two types of pointers: links that are openly visible to the attacker upon reading the files (as it was in level1), and

Fig. 2 Example of webserver at level1. Nodes represent files and solid arrows represent connections between files

Table 1 Actions in level1

Action name	Parameters	Result
$\text{read}()$	File	Set of files
$\text{search}()$	File	True/false

Table 2 Number of actions and states in level1

Number of files	Number of actions	Number of states
N	$2N$	2^{2N-1}
2	4	8
3	6	32
5	10	512
10	20	$\approx 5 \times 10^5$

implicit pointers that requires an actual analysis of the file. Real-world examples of these types of implicit pointers may be: comments in the source code that refers to another file without stating a direct link; keywords used in the file that refer to a special type or version of a webserver app or CMS, and that indicate the existence of other default files; recurrent appearance of a word, suggesting that there may be a file or folder with the same name. Practically, level2 problems can be represented as directed typed graph of files with two types of edges (see Fig. 3).

The set of actions of the agent is now extended to three parametric actions $\mathcal{A} = \{\text{read}(\text{file}_i), \text{search}(\text{file}_i), \text{deepread}(\text{file}_i)\}$. As before, action $\text{read}(\text{file}_i)$ reads the i^{th} file and returns a list of files connected by an *explicit* link, while $\text{search}(\text{file}_i)$ checks the i^{th} file for the presence of the flag. The action $\text{deepread}(\text{file}_i)$ processes the i^{th} file and returns a list of files connected by *implicit* links. See Table 3 for a summary of the actions, their parameters, and their return values. Notice that at this level of abstraction, the logic and the algorithm for performing a $\text{deepread}()$ are implicitly provided in the game itself. At higher levels of abstraction, the task of actually parsing an HTML file and uncover the possible URLs of new files would be delegated to the learning agent; such an agent would receive the actual content of a file and it could use a range of algorithms to process the text, from simple dictionary mapping (e.g., *apache*

Fig. 3 Example of webserver at level2. Nodes represent files, solid arrows represent direct connections, and dashed arrows represent indirect connections between files

Table 3 Actions in level2

Action name	Parameters	Result
read()	File	Set of files
deepread()	File	Set of files
search()	File	True/false

mapping to *cgi-bin*, *wordpress* mapping to *wp-login*, etc.) to more complex natural language processing neural networks able to propose new potential file candidates.

Given N files on the webserver, the cardinality of the action space is now $|A| = 3N$ and the cardinality of agent state space is 2^{3N-1} , by trivially scaling up from level1 because of an additional action. Table 4 shows estimates for few values of N .

4.2.3 Level3: dynamic content layer

The real complexity of a website starts with server-side scripting. In level3 we consider a webserver that can dynamically execute server-side scripts by processing user parameters and generating static content for the client. We still model the webserver as a collection of static files, delegating the complexity of dynamic server-side scripting in the space of actions. From a practical perspective, the webserver can still be seen as directed typed graph with nodes that may return different values depending on the received parameter (see Fig. 4).

The size of the action space remains constant, but in order to account for parameter passing, we now redefine the signature of the actions to include new parameters: $\mathcal{A} = \{\text{read}(\text{file}_i, \text{pname}_j, \text{pval}_k), \text{search}(\text{file}_i, \text{pname}_j, \text{pval}_k), \text{deepread}(\text{file}_i, \text{pname}_j, \text{pval}_k)\}$. Actions have the same semantics as in level2, but now, beyond receiving file i as an input parameter, they also receive parameter name j and parameter value k . This reflects the request of a specific URL (file i) together with a specific parameter (parameter name j) and a set value (parameter value k). The return value of the *read()* and *deepread()* actions is also enriched by a possible set of parameter names and values; this is due to the fact that the answer of the webserver may contain not only links

Table 4 Number of actions and states in level2

Number of files	Number of actions	Number of states
n	$3N$	2^{3N-1}
2	6	32
3	9	256
5	15	16384
10	30	$\approx 5.3 \times 10^8$

Fig. 4 Example of webserver at level3. Solid nodes represent files, dotted nodes within a file illustrate a pair of parameter name and value that may be sent to a file, solid arrows and dashed arrows represent, respectively, direct and indirect connections between files given a parameter pair. If an arrow leads to a file, it means that upon a successful *read()* or *deepread()* action the file itself is revealed without parameters; if an arrow leads to an internal dotted node, then after a successful *read()* or *deepread()*, a file together with a parameter list for the file is also sent back to the agent

to other files, but it may include the specific parameter pairs relevant to the connected files. See Table 5 for a summary of the actions, their parameters, and their return values. Notice that at this level of abstraction, we assume that only a single pair $(\text{pname}_j, \text{pval}_k)$ can be specified as input; moreover, to keep the complexity in check, we assume that pname_j and pval_k may assume values in a finite set, that is, $1 \leq j \leq M$ and $1 \leq k \leq O$, $M, O \in \mathbb{N}_{\geq 0}$.

The cardinality $|\mathcal{A}|$ of the action space is now much larger because of combinatorial explosion in the parameters of an action. Assuming N files on the webserver, and a set of M parameter names and O parameter values that can be freely combined, each action can be instantiated $N + NMO$ times (N times without parameters, and NMO times considering

Table 5 Actions in level3

Action name	Parameters	Result
read()	File	Set of files
	Parameter name	Set of parameter
	Parameter value	Names and values
deepread()	File	Set of files
	Parameter name	Set of parameter
	Parameter value	Names and values
search()	File	True/false
	Parameter name	
	Parameter value	

all combinations). In total, we then have $3(N + NMO)$ concrete actions the agent can take. A trivial agent that explicitly tracks its state by recording, for each possible action, if it was taken or not, would have to deal with a space with a cardinality of $2^{3(N+NMO)-1}$. Table 6 shows some estimates for different values of N , M , and O .

4.2.4 Level4: web method layer

In level4, we further scale the complexity by considering the possibility of a webserver receiving a request specifying a HTTP web method and containing a list of parameter names and parameter values. The webserver is always modeled as a collection of files forming a directed typed graph with nested nodes (see Fig. 5).

The set of parametric actions is now restructured. We drop the previous artificial distinction between `read()`, `deepread()`, while in previous levels of abstraction the task of extracting explicit and implicit links was exter-

nalized in the logic of the game, from now on it is the task of the agent to parse and analyze the answer of the webserver in order to find explicit and implicit links, as well as the flag itself. The new action set is: $\mathcal{A} = \{\text{get}(\text{file}_i, [\text{pname}], [\text{pval}]), \text{post}(\text{file}_i, [\text{pname}], [\text{pval}])\}$. The `get()` and `post()` actions implement the respective web methods, and they receive as input a file (file_i), a list of parameter names ($[\text{pname}]$) together with a list of parameter values ($[\text{pval}]$). The result of these actions is an HTTP page. The flag is considered retrieved when the agent obtains the HTTP page containing the flag. See Table 7 for a summary of the actions, their parameters, and their return values.

The aim of the level4 abstraction is to consider dynamic website content based on multiple dynamic parameter combinations sent by the client in different ways. This is a more advanced abstraction of the problem compared to level3, where the files accepted only one dynamic parameter without specifying the way how it was sent. Notice that, on the other hand, HTTP protocol is capable of carrying out many additional operations such as testing the path to target with the *TRACE* method, or receiving answers without the response body with *HEAD*. These methods have no additional value in level4, since the aim is to capture the dynamic response body. Other methods enable modifying the website content by creating objects with the *PUT* method, or removing objects with the *DELETE*; however, these operations are only considered in higher layers in the Agent Web Model. In this sense, the name web method layer can be misleading, but we chose it because in most of the cases the *GET* and the *POST* methods are the most used in web communication.

Given, as before, N files on the webserver, M possible alternatives for the parameter names, O possible alternatives for the parameter values, the cardinality $|\mathcal{A}|$ depends on the maximum length P of the list of parameters. With $P = 0$, $|\mathcal{A}| = 2N$, that is, trivially, `get()` and `post()` actions with no parameter on each file. With $P = 1$, $|\mathcal{A}| = 2N + 2NMO$, that is, the same two actions for every possible combination of zero or one parameter name and value (similar to level3). In the worst case in which $P = M$, that is the list can be long enough to contain all the parameter names, the number of possible actions could be estimated as:

Table 6 Number of actions and states in level3

#files	#pars	#pvals	#actions	#states
N	M	O	$3(N + NMO)$	$2^{3(N+NMO)-1}$
2	2	2	30	$\approx 5.4 \times 10^8$
2	5	5	156	$\approx 4.6 \times 10^{46}$
5	2	2	75	$\approx 1.9 \times 10^{22}$
5	5	5	390	$\approx 1.3 \times 10^{117}$
10	5	5	780	$\approx 3.2 \times 10^{234}$

Fig. 5 Example of webserver at level4. Solid nodes represent files, dotted nodes within a file illustrate possible lists of parameter names and values that may be sent to a file via a webmethod, solid arrows represent connections between files given parameters. Inside the internal nodes the used webmethod is indicated

Table 7 Actions in level4

Action name	Parameters	Result
get()	File	HTTP page
	Set of parameter names	
	Set of parameter values	
post()	File	HTTP page
	Set of parameter names	
	Set of parameter values	

$$\underbrace{2}_{\text{actions}} \cdot \underbrace{N}_{\text{files}} \cdot \underbrace{\sum_{i=0}^M}_{\text{all list lengths}} \cdot \underbrace{\binom{M}{i}}_{\text{all combinations of } i \text{ param names}} \cdot \underbrace{O^i}_{\text{all combinations of } i \text{ param vals}}$$

A trivial agent that would store again its state knowledge about actions using binary values would have to deal with a state space of cardinality $2^{|A|}$.

4.2.5 Level5: HTTP header layer

While all the previous layers considered only the URL and the body part of the HTTP packets, level5 takes the HTTP header into consideration as well. The HTTP header can contain relevant information such as the session variables or the web response code in the response header. The session, which is composed of a session variable name and value (e.g., *JSESSIONID=Abvhj67*), is used to provide elevated access to special users; a practical example is the login process (which may happen by sending multiple *POST* parameters, as modeled in level4), after which the server sets a new session value. Additional HTTP header information, such as the browser type or the character encoding, can also have an effect on the response provided by the webserver.

We always model the webserver as a collection of files forming a directed typed graph with nested objects (see Fig. 6). Object access now depends also on the header variables. We consider pairs of session name and session value as a single parameter (session values are usually random numbers with high entropy so there is no point in handling the session variable name and value separately unless the session values are predictable and the attacker wants to brute-force the session value), and we limit the number of allowed session pairs and HTTP headers. Under this assumption, we preserve the same actions as level4, but we extend the signature of their input parameters: $\mathcal{A} = \{\text{get}(\text{file}_i, [\text{pname}], [\text{pval}], [\text{sess}], \text{header}), \text{post}(\text{file}_i, [\text{pname}], [\text{pval}], [\text{sess}], \text{header})\}$. Beside receiving an input file (file_i), a list of parameter names ($[\text{pname}]$) together with a list of parameter values ($[\text{pval}]$), the *get()* and *post()* methods now also receive a list of session pairs ($[\text{sess}]$) and an HTTP header (*header*). The result of these actions is a web response, possibly together with an HTTP page. The web response code (e.g., 200, 404, 500) reflects the accessibility of the requested object. As before, the flag is considered retrieved when the agent obtains the HTTP page containing the flag. See Table 8 for a summary of the actions, their parameters, and their return values.

Fig. 6 Example of webserver at level5. Solid nodes represent files, dotted nodes within a file illustrate possible lists of parameter name and value pairs and session name and value pairs that may be sent to a file via a webmethod, solid arrows represent, respectively, connections between files given parameters and sessions

Table 8 Actions in level5

Action name	Parameters	Result
get()	File	HTTP page
	Set of parameter names	Web response
	Set of parameter values	
	Set of session pairs	
	HTTP header	
post()	File	HTTP page
	Set of parameter names	Web response
	Set of parameter values	
	Set of session pairs	
	HTTP header	

With reference to the actions we have defined, we observe an enlargement of the action space that now depends on the number N of files on the server, the number M of parameter names that can be selected, the number O of parameter values available, the number P of parameter pairs that can be sent, the number Q of session pair values available, the number R of session pairs that can be sent, and the number S of HTTP header without cookies that can be sent.

Figure 6 provides also the illustration of a possible interaction between the agent and the webserver. The attacker first tries to log in using an invalid password, which actually reveals a new version of the *login.php* file by redirecting the page to the *index.php* page without session. Using the right credentials shows another version of the *login.php* page that instead redirects the user to a version of *index.php* with the

session pair *sessionpair1*. This version of the *index.php* leads then to another version of the file (logout action) that is connected to the original version of *index.php* without session.

4.2.6 Level6: server structure layer

In a complex web hacking scenario, the attacker may map the file system of the server in order to collect information to be used during the attack. In level6, we extend the formalization of the webserver in order to consider not only files within the webroot, but also objects beyond it, such as local files and databases. This extension allows to simulate attacks relying on local file inclusion (LFI) vulnerabilities, or information gathering attacks on a database in order to set up a SQL injection. Figure 7 shows the structure of a webserver, and it illustrate a possible LFI attack to obtain the webserver logs or the environmental variables. Level6 abstraction provides the agent the following additional features compared to lower level of abstractions:

- Obtaining the local resources of the website such as the background files
or the background database records used for the website operation.
- Accessing the data in order to compromise other websites residing on the same webserver;
- Obtaining the webserver files that are used for other purposes than the website operations, such as users data or operating system data

In this scenario, the access rights of the objects play an important role; running a webserver as a root can have serious consequences, while having minimum access rights reduce the chance of such exploitations.

While the action set remains the same as level5, the extension of the domain of the objects beyond the webroot escalates the number of targets that the agent may consider. Complexity soars with the increase of objects, including databases, and, within a database, its tables, columns and rows.

4.2.7 Level7: server modification layer

The last level we consider in our Agent Web Model is the server modification layer. In this level, we assume that the agent can carry out complex meaningful web hacking actions such as creating its own objects, either inside or outside the web root. With the ability to create its own files, the attacker can place command scripts that can be used to carry out advanced attacks. Figure 7 shows the same structure of the server as in level6, and it illustrates an attacker creating its own files on the webserver. Level7 abstraction provides the

Fig. 7 Example of webserver at level6. Solid nodes represent files, dotted nodes within a file illustrate possible lists of parameter name and value pairs and session name and value pairs that may be sent to a file via a webmethod, solid arrows represent connections between files given parameters and sessions. Dotted boundary lines separate different logical spaces, such as the webserver space and the database space. Dashed arrows mark connections between these logical spaces

agent the following additional features compared to lower level of abstractions:

- Causing denial of service by editing important objects for the site operation;
- Defacing the site by changing the site content;
- Escalating privileges by adding data to objects;
- Uploading attack scripts to provide extra functions for the attack;
- Removing attack clues by deleting log files, deleting temporary files that were used for the attack.

Attacking actions leading to the creation of objects can be carried out by the web requests that we have already considered. The action does not change, but the domain of the parameters increases in order to allow for more sophisticated actions.

Level7 is assumed to be the highest level of modeling, capturing all relevant features of hacking; thus, solving this challenge is extremely hard, and we would expect that a successful agent would perform as well as, or better than, a professional human hacker actually involved in a process of website hacking.

Fig. 8 Example of webserver at level7. Solid nodes represent files, dotted nodes within a file illustrate possible lists of parameter name and value pairs and session name and value pairs that may be sent to a file via a webmethod, solid arrows represent connections between files given parameters and sessions. Dotted boundary lines separate different logical spaces, such as the webserver space and the database space. Dashed arrows mark connections between these logical spaces. Boldface objects represent objects created by the attacker

5 Modeling web vulnerabilities

In this section, we analyze how different types of web vulnerabilities fit within our *Agent Web Model*. For each vulnerability, we present the minimal requirements for the presence of the vulnerability and different possible exploitation strategies. We then discuss at which level of the Agent Web Model hierarchy these vulnerabilities may be modeled, and how the parameters of the Agent Web Model can be used to express the actual parameters needed for exploitation (e.g., how the HTTP header information or objects outside the webroot can be mapped to parameters in the Agent Web Model). Table 9 offers a summary of all the vulnerabilities, together with the level of the Agent Web Model at which they can be modeled.

Information disclosure is a type of vulnerability where the attacker gains useful information by penetrating the system. Evaluating the usefulness of the gained information is not trivial, but through the CTF formalization we make the simplifying assumption that relevant information the attacker may be interested into is marked by the flag. In this way, it is possible to equate successful information disclosure with the retrieval of the flag. Every level of abstraction in our *Agent Web Model* captures this attack: in level1 sensitive information (flag) is in a public linked file on the webserver; in level2 sensitive information (flag) can be inside a private file; in the

Table 9 Web vulnerabilities in the Agent Web Model

Agent Web Vulnerability Layers	Web Model	Comment
Information	1–6	Flag in web files
Disclosure		Header or outside object
Parameter	3–5	Flag access with modified
Tampering		Parameters or sessions
Cross site	3–5	Flag access with
Scripting		Client-side redirection
Cross site	5	Only CSRF token
Request forgery		Manipulation
SQL injection	3–7	Flag with authentication Bypass or from database
Xpath injection	3–4	Flag with authentication Bypass or from XML file
Server-side	3–7	Flag with object access
Template injection		Or privilege escalation
Remote file	3	Flag with remote
Inclusion		File access
Local file	3–7	Flag with local
Inclusion		File access
Session related	5–6	Flag with session
Attacks		Manipulation
HTTP response	5	Flag with HTTP header
Splitting		Manipulation

following layers (level3 to level5) sensitive information (flag) can be accessed using special parameters or sessions; in level 6, sensitive information (flag) can be inside a file outside the webroot.

Web parameter tampering [19] is a type of attack where the web parameters exchanged by the client and the server are modified in order to have access to additional objects. Our *Agent Web Model* captures this attack starting at level3 by allowing the specification of web parameters in the URL; in level4 it is possible to add HTTP body parameters (POST message); in level5 it is possible to edit cookies in the HTTP header. In all these instances, an agent can perform web parameter tampering either by meaningfully exploring the space of possible values of these parameters, or by trying to brute-force them.

Cross site scripting (XSS) attacks [16] enable attackers to inject client-side (e.g., JavaScript) code into the webpage viewed by other users. By exploiting a XSS vulnerability, the attacker can overwrite the page content on the client side, redirect the page to the attacker's page, or steal the valid sessions inside the cookie. All these offensive actions can be followed by some social engineering trick in case of a

real attack. In the context of CTF style challenges where additional clients are not available, the aim of an attacker is simply to show the existence of the vulnerability. A flag may be used to denote a page that is only accessible indirectly by redirection. The task for the agent is to find the right parameters to achieve the redirection. The injected client-side code for XSS has to be sent through web parameters. XSS attacks can be simulated in our *Agent Web Model* as soon as we can interact with parameters: in level3 the attacker may add code in the URL; in level4 the attacker may modify POST parameters; in level5 the XSS attack may affect the header.

Cross site request forgery (CSRF) [36] is a type of vulnerability where the attacker sends a link to authenticated users in order to trick them to execute web requests by social engineering. If the users are authenticated (have sessions), the malicious request (e.g., transferring money, changing the state) is executed by the server. This exploitation is based on social engineering and on misleading the user. In addition, CSRF tokens are sent by the server to filter out unintended requests; the agent can check the existence of appropriate CSRF tokens or exploit requests with weak CSRF tokens. In our model, the CSRF attack has to be simplified to consider only the CSRF token manipulation in level5.

SQL injection [1] is a vulnerability where malicious SQL statements can be executed by the server due to the lack of input validation on the server side. By modifying the original SQL statement of a server-side script, the attackers can bypass authentication, access confidential database information or even write attack scripts on the server (select into outfile command). In most of the cases, the attacker has to map the database structure of the target by finding, for instance, the different table names along with their column names and types. In our *Agent Web Model*, this attack can be completely simulated at level6 (where we consider the existence of objects outside the webroot), although other simplified versions may happen at lower levels. In the easiest case, the agent only need one dynamic parameter without sessions; bypassing only a simple authentication or collecting data from the same table that the server-side script uses does not require to know the table name and other database structure data; in these cases, a basic form of SQL injection may be simulated even in level3 (with one vulnerable parameter). Complex cases comprising all the database parameters need to happen at level6. If the attacker uses the SQL injection to carry out further actions such as writing attacking scripts on the compromised site, then this has to happen at level7, where the agent can modify server by creating files. All the above-mentioned cases require a very high number of actions especially when the agent has to execute a Boolean-based blind SQL injection. In these cases, the vulnerable application provides only true or false answers, so obtaining one single piece of information, such as a column name in a

table, requires binary search type requests for each character, which can lead to an exponential number of actions. Notice that the Agent Web Model abstraction does not consider the response time of the environment. In very specific cases such as time-based blind SQL injections, the attacker may have to measure the response time; this type of exploitation would require the consideration of the server reaction time too.

Xpath injection [5] is a web vulnerability where the attacker injects code into the web request, but the target of the attack is not a database (as in the case of SQL injection) but an XML file. By exploiting Xpath injection, the attacker can iterate through XML elements and obtain the properties of the nodes one by one. This operation requires only one parameter, so simulating Xpath injection is theoretically possible in level3. Since the exploitation of the Xpath injection does not require the name of the XML file, mapping the files outside the webroot is not necessary even if the XML file is outside the webroot. On the other hand, the vulnerable parameter can be a POST parameter (level4) or it can require a specific session (level5).

Server-side template injection (SSTI) [21] is a vulnerability where the attacker uses native template syntax to inject a malicious payload into a website template. For the exploitation the agent has to use additional actions that are SSTI specific, such as sending a string like `${7*7}` together with a parameter. Theoretically, an easy SSTI vulnerability can be exploited in level3, but all other layers above can be used to represent specific attack cases (vulnerable parameter in POST on level4, session required for exploitation on level5); in particular cases, the attacker can list the server structure (level6) or can create files with arbitrary code execution (level7).

File inclusion [20] makes the attacker capable of including remote or local files by exploiting a vulnerable web parameter on the website. In case of remote file inclusion (RFI), the attacker can include its own remote attacking script in the server-side script. Remote file inclusion can have very serious consequences, but in a CTF challenge the aim is just to show the possibility of the exploitation, not to carry out an actual exploit. RFI can be realized by providing a remote file that sends the flag if the request is initiated from the target website IP. Exploiting RFI is possible in level3 but other parameters, such as POST request and sessions, can be relevant (level4 and level5). As a consequence of the RFI vulnerability the attacker can create files on the website for further attacks. In case of local file inclusion (LFI), the attacker can include local files in the server-side script. For the exploitation one single parameter is theoretically enough, but since usually it is necessary to read local files outside the webroot, the agent has to map at least a part of the server structure (level6). In some exploitation scenarios the attacker can use local files (such as logs or files in the */proc* Linux folder) to create its own files on the server (level7).

Session-related attacks [43] exploit session disclosure or other weaknesses in the session generation process. Since we model the environment as the server itself without other network nodes, man in the middle session disclosures cannot be considered. Other session disclosures can be possible, for instance, if the sessions are stored in the logs and the website can access the log files (LFI), as modeled on level6. Brute-forcing the session is also possible in level5, but brute-force actions increase dramatically the complexity and the number of possible actions.

HTTP response splitting [18] is a vulnerability where the attacker can control the content of the HTTP header of a web request. The ability of the attacker to construct arbitrary HTTP responses can result in many other exploits such as cache poisoning or Cross Site Scripting. Our *Agent Web Model* considers the HTTP header information in level5, but only with limited information (different session pairs and the whole header together with different versions). Training the agent to learn HTTP response splitting exploitation would require to split the HTTP header in multiple parts and allow the agent to consider actions on different HTTP header combinations.

6 Implementation of the Agent Web Model

An implementation of the first three levels of the Agent Web Model has been developed in agreement with the standard defined in the *OpenAI gym* framework [7], and it has been made available online.¹ By adopting the standardized *OpenAI gym* interface, we hope to make it easy for researchers and practitioners to test their agents and algorithms on CTF challenges. In particular, we hope to simplify the process of deploying and training off-the-shelf RL agent, as well as provide interesting problems that may promote the development of new learning algorithms.

In our implementation, each level defines a simple interface to an abstraction of a CTF challenge. The environment is given by a webserver instantiated as an *OpenAI gym* object which makes available to the agent a finite set of actions. Actions taken by the agent are processed through a `step()` method that returns the outcome of the action, a reward, a termination signal, and an optional debug message. Environments at different levels of abstraction may be instantiated parametrically (deciding the number of files, the links, and the possible parameters), thus offering the possibility of generating a wide variety of challenges for a learning agent.

The implementation of the first level provides a simple, tutorial-like, CTF game. The constructor `env(A, flag)` of the CTF challenge receives an adjacency matrix *A* for the files on the server, and an integer `flag` location of the flag; it then

instantiates the webserver in the form of a directed graph (see Fig. 9 for the actual implementation of the logical webserver shown in Fig. 2). Actions are exposed in the form of dictionaries with two arguments: an integer `command` determining the type of action to be taken (corresponding to the column *action name* in Table 1) and an integer `targetfile` specifying on which file the action is taken (corresponding to the column *parameters* in Table 1). Responses from the webserver follow a standard formatting where the outcome argument is either a Boolean value or a list of integers denoting files (in accordance with the column *result* in Table 1).

The second level builds over the first one, introducing an additional type of connection between the files on the webserver. The constructor `env(A, B, flag)` now receives two adjacency matrices *A* and *B*, the first encoding direct connections and the second encoding indirect connections. Actions preserve the same syntax, although now the `command` constant accepts one more value corresponding to the `deepread()` action (see Table 3). Responses keep the same form as in level1.

Finally, the third level constitutes a non-trivial abstraction of a real hacking challenge, where we consider a webserver that interacts with the actions of the attacker in specific ways. The constructor `env(n_files, n_pnames, n_pvalues, webserver)` now receives the number of files on the webserver *n_files*, as well as the number of available parameter names *n_pnames* and values *n_pvalues*; finally, the constructor receives a function `webserver()` which is called in the `step()` function and which is tasked with processing the actions of the attacker according to its own logic. Notice that, at this level, it is not necessary to input an explicit adjacency matrix for the files anymore; the

Fig. 9 Example of implementation of webserver at level1

¹ <https://github.com/FMZennaro/gym-agentwebmodel>.

internal structure of the webserver is encoded in the function `webserver()` itself. Actions are still dictionaries, as in `level2`, with two additional integer arguments, `pname` and `pvalue`, thus complying with the definition in column *parameters* of Table 5. The generated responses are shaped in the same form as in `level2`, thus returning either a Boolean or a set of integers denoting files (notice that, comparing with the *result* column in Table 5, we avoid explicitly returning parameter values and parameter names referring back to action input).

Even these simple challenges may be already seen as simple models of real-world vulnerability. For instance, with reference to Sect. 5, `level1` and `level2` simulations allow us to model simple information disclosure vulnerabilities on web sites; `level3` allows us to model several easy vulnerability exploitation, such as SQL injection (e.g., bypassing SQL-based website login via SQL injection with specific parameter values) or file inclusion (e.g., reading the source via local file inclusion with specific parameters).

To validate our framework, we test it by deploying RL agents from the *stablebaselines*² library. We train and evaluate synchronous advantage actor-critic (A2C) agents [27] using off-the-shelf configuration on the three levels presented above. In particular, we set up `level1` with seven files; `level2` with eleven files; and `level3` with four files, five parameter names and five parameter values. In all the levels, the position of the flag is randomized at the beginning of the simulation; a reward of 100 is given for retrieving the flag, and a reward -1 for any other action.

Figure 10 shows the dynamics of learning. The standard agents were able to interface themselves with the implementations of CTF challenges at different levels of abstractions. All the agents were able to learn: the smoothed long-term trajectory shows an increase in the final reward achieved by the agent (notice that the initial high variance is due to the absence of historical data for smoothing). However, the quality of learning varies sensibly with the level of the simulation: for `level1` the agent quickly approaches an optimal solution, while for `level3` the final reward is very low (notice the negative value on the y-axis); in this last case, although the agent learns, it is far from an optimal policy.

On one hand, these simulations show the feasibility of defining CTF challenges using a standard interface which would allow for the quick and easy deployment of RL agents. On the other hand, they also highlight issues of feasibility related to the use of RL algorithms; solving a problem at `level3` and beyond requires either computational resources or more refined learning agents. A core strength of the Agent Web Model is to provide a standardized paradigm for the researcher in computer security to model a wide array of security challenges on the web, as discussed in Sect. 5. This

Fig. 10 Reward collected by agents as a function of the number of training episode in `level1` (orange line), `level2` (green line), and `level3` (pink line). The solid line represents the smoothed reward; the shaded line represents the actual rewards

paradigm would help the modeler in evaluating at what level of abstraction to represent a problem of interest, and it would provide a versatile interface to the world of RL. In addition to this, the problem decomposition enabled by the Agent Web Model can help security researchers to focus on simpler and smaller and introduce the practical application of RL step by step in ethical hacking. In sum, the Agent Web Model framework may provide both a resource for researcher in computer security to model their problems and tackle them using standard agent, and an inspiration for researchers in machine learning to develop new algorithms able to tackle the hard challenges of CTF games.

7 Ethical considerations

RL agents trained for ethical penetration testing carry with them the potential for malicious misuse. In particular, the same agents may be deployed and adapted with the aim of generating material or immaterial damage. We would like to repeat that the aim of the current study is to develop agents to assist ethical hackers in legitimate penetration testing, and to develop an understanding of RL agents on a preventive ground only. For this reason, we advocate the development of agents in the context of CTF challenges, where the aim is a minimal and harmless exploitation of a vulnerability as a proof-of-concept (capture of the flag), but no further attacks are considered. We distance ourselves and condemn any application of these results for the development of offensive tools, especially in a military context.³

8 Conclusions

In this paper, we presented a model, named *Agent Web Model*, that defines web hacking at different levels of abstraction. This formulation allows for a straightforward implementation of problems suited for machine learning agents. Since the aim and type of web attacks can be various, and different technical and human methods may be involved, we first restricted

² <https://stable-baselines.readthedocs.io/en/master/>.

³ <https://futureoflife.org/open-letter-autonomous-weapons/>.

our attention to CTF-style hacking problems. We then modeled CTF-style web hacking as a game and as a RL problem. The RL problem considers a single player dealing with a static website consisting of objects with which the agent can interact by sending requests (with or without parameters). We formalized RL problems on 7 different levels of abstraction, ordered by increasing complexity in terms of number of objects, actions, parameters and states. Starting from a simple challenge on the first level of abstraction, we observed the complexity of the problems quickly increasing, thus defining a non-trivial learning challenge for an artificial agent. An implementation of the problems on the first levels of abstraction was provided. The challenges we implemented range in complexity, they allow for customizability, and provide a way to instantiate a large number of random web hacking challenges in a generative way in order to train an artificial agent. Finally, we showed how these implementations may be readily tackled by deploying off-the-shelf RL agents. Other real-world security challenges may be analogously modeled, and future work will be directed to further developing and further standardizing CTF challenges at higher levels of abstraction, as well as applying state of the art RL techniques to the problems we defined. It is our hope that the formalization presented in this paper may not only allow for the development of automatic red bots that may help in the task of ethical penetration testing, but also promote the interaction and the research in both fields of machine learning and computer security: helping security expert to define realistic and relevant challenges that meet the formalism of machine learning, and offering to the RL expert stimulating problems that may foster advances in machine learning.

Funding Open access funding provided by University of Oslo (incl Oslo University Hospital).

Declarations

Conflict of interest All authors declare that they have no conflict of interest.

Human and animals rights This article does not contain any studies with human participants or animals performed by any of the authors.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Anley, C.: Advanced SQL injection in SQL server applications. https://crypto.stanford.edu/cs155old/cs155-spring11/papers/sql_injection.pdf. Accessed: 2020-09-12 (2002)
2. Antunes, N., Vieira, M.: Designing vulnerability testing tools for web services: approach, components, and tools. *Int. J. Inf. Secur.* (2017)
3. Appelt, D., Nguyen, C.D., Panichella, A., Briand, L.C.: A machine-learning-driven evolutionary approach for testing web application firewalls. *IEEE Trans. Reliab.* **67**(3) (2018)
4. Berners-Lee, T., Fischetti, M.: *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. DIANE Publishing Company, Darby (2001)
5. Blasco, J.: Introduction to XPath injection techniques. <http://repository.root-me.org/Exploitation%20-%20Web/EN%20-%20Introduction%20to%20XPath%20injection%20techniques.pdf>. Accessed 2020-09-12 (2007)
6. Boddy, M.S., Gohde, J., Haigh, T., Harp, S.A.: Course of action generation for cyber security using classical planning. In: *ICAPS*, pp. 12–21 (2005)
7. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym. *arXiv preprint arXiv:1606.01540* (2016)
8. Chowdary, A., Huang, D., Mahendran, J.S., Romo, D., Deng, Y., Sabur, A.: Autonomous security analysis and penetration testing. In: *The 16th International Conference on Mobility, Sensing and Networking (MSN 2020)* (2020)
9. Damele, B., Stampar, M.: sqlmap user's manual. <http://www.it-docs.net/ddata/4956.pdf>. Accessed 2020-05-09 (2011)
10. Elderman, R., Pater, L.J., Thie, A.S.: Adversarial reinforcement learning in a cyber security simulation. PhD thesis, Faculty of Science and Engineering (2016)
11. Fielding, R., Irvine, U.C., Gettys, J., Mogul, J., Frystyk, H.: Request for comments 2616, hypertext transfer protocol – http 1/1. <https://tools.ietf.org/html/rfc2616>. Accessed: 2020-09-05 (1999)
12. Fonseca, J., Vieira, M., Madeira, H.: Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. In: *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)* (2020)
13. Frazee, D.: Cyber grand challenge (CGC). <https://www.darpa.mil/program/cyber-grand-challenge>. Accessed 2020-05-09 (2016)
14. Ghanem, M.C., Chen, T.M.: Reinforcement learning for intelligent penetration testing. In: *2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)* (2018)
15. Ghanem, M.C., Chen, T.M.: Reinforcement learning for efficient network penetration testing. *Information* **11**(1), 6 (2020)
16. Grossman, J., Hansen, R., Petkov, D., Rager, A., Fogie, S.: *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress (2007)
17. Hoffmann, J.: Simulated penetration testing: From “Dijkstra” to “Turing test++”. In: *Twenty-Fifth International Conference on Automated Planning and Scheduling* (2015)
18. Howard, M., LeBlanc, D., Viega, J.: 24 deadly sins of software security, sin 2: Web server-related vulnerabilities (XSS, XSRF, and response splitting). <http://index-of.es/Miscellaneous/24-DEADLY-SINS-OF-SOFTWARE-SECURITY-2010.pdf>. Accessed 2020-09-12 (2010)
19. Howard, M., LeBlanc, D., Viega, J.: 24 deadly sins of software security, sin 4: Use of magic URLs, predictable cookies, and hidden form fields. <http://index-of.es/Miscellaneous/24-DEADLY-SINS-OF-SOFTWARE-SECURITY-2010.pdf>. Accessed 2020-09-12 (2010)
20. Johnson, G.: Remote and local file inclusion explained. <http://repository.root-me.org/Exploitation%20-%20Web/EN%20-%20Remote%20and%20local%20file%20inclusion%20explained.pdf>. Accessed 2020-09-12 (2007)

- [%20Remote%20File%20Inclusion%20and%20Local%20File%20Inclusion%20explained.pdf](#). Accessed 2020-09-12 (2008)
21. Kettle, J.: Server-side template injection: RCE for the modern webapp. <https://www.blackhat.com/docs/us-15/materials/us-15-Kettle-Server-Side-Template-%Injection-RCE-For-The-Modern-Web-App-wp.pdf>. Accessed 2020-09-12 (2015)
 22. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: *Advances in Neural Information Processing Systems*, pp. 1097–1105 (2012)
 23. Lison, P., Mavroeidis, V.: Automatic detection of malware-generated domains with recurrent neural models. *arXiv preprint [arXiv:1709.07102](#)* (2017)
 24. Maeda, R., Mimura, M.: Automating post-exploitation with deep reinforcement learning. *Comput. Secur.* **100**, 102108 (2021)
 25. McDaniel, L., Talvi, E., Ba, H.: Capture the flag as cyber security introduction. In: *Annual Hawaii International Conference on System Sciences (HICSS)* (2016)
 26. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. *Nature* **518**(7540), 529–533 (2015)
 27. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: *International Conference on Machine Learning*, PMLR, pp 1928–1937 (2016)
 28. Niculae, S., Dichiu, D., Yang, K., Bäck, T.: Automating penetration testing using reinforcement learning. <https://stefann.eu/files/Automating%20Penetration%20Testing%20using%20Reinforcement%20Learning.pdf> (2020)
 29. Pettersson, A., Fjordefalk, O.: Using Markov decision processes and reinforcement learning to guide penetration testers in the search for web vulnerabilities (2019)
 30. Pozdniakov, K., Alonso, E., Stankovic, V., Tam, K., Jones, K.: Smart security audit: reinforcement learning with a deep neural network approximator. In: *2020 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, pp. 1–8 (2020)
 31. Rogers, R.: *Nessus Network Auditing*, 2nd edn. Syngress, Burlington (2011)
 32. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated vulnerability detection in source code using deep representation learning. In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, pp. 757–762 (2018)
 33. Sarraute, C., Buffet, O., Hoffmann, J.: Penetration testing== POMDP solving? *arXiv preprint [arXiv:1306.4714](#)* (2013)
 34. Schwartz, J., Kurniawati, H.: Autonomous penetration testing using reinforcement learning. *arXiv preprint [arXiv:1905.05965v1](#)* (2019)
 35. Shalev-Shwartz, S., Shammah, S., Shashua, A.: Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint [arXiv:1610.03295](#)* (2016)
 36. Siddiqui, S., Verma, D.: Cross site request forgery: a common web application weakness. In: *International Conference on Communication Software and Networks, ICCSN* (2011)
 37. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of Go without human knowledge. *Nature* **550**(7676), 354 (2017)
 38. Speicher, P., Steinmetz, M., Hoffmann, J., Backes, M., Künnemann, R.: Towards automated network mitigation analysis. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pp. 1971–1978 (2019)
 39. Stasinopoulos, A., Ntantogian, C., Xenakis, C.: Commix: automating evaluation and exploitation of command injection vulnerabilities in web applications. *Int. J. Inf. Secur.* (2019)
 40. Sutton, R.S., Barto, A.G.: *Reinforcement Learning: An Introduction*. MIT Press, Cambridge (2018)
 41. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: *Advances in Neural Information Processing Systems*, pp. 5998–6008 (2017)
 42. Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P., et al.: Grandmaster level in Starcraft II using multi-agent reinforcement learning. *Nature* **575**(7782), 350–354 (2019)
 43. Vlsaggio, C.A., Blasio, L.C.: Session management vulnerabilities in today's web. *IEEE Security and Privacy* (2010)
 44. Wichers, D., Williams, J.: Owasp - top 10 web application security risks. <https://owasp.org/www-project-top-ten/>. Accessed 2020-09-05 (2017)
 45. Zennaro, F.M., Erdodi, L.: Modeling penetration testing with reinforcement learning using capture-the-flag challenges and tabular Q-learning. *arXiv preprint [arXiv:2005.12632](#)* (2020)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.