Automating XSS Vulnerability Testing Using Reinforcement Learning

Kento Hasegawa[a], Seira Hidano and Kazuhide Fukushima[b]
KDDI Research, Inc., 2-1-15, Ohara, Fujimino, Saitama, Japan

Keywords: Cross-Site Scripting, Reinforcement Learning, Vulnerability Testing.

Abstract: Cross-site scripting (XSS) is a frequently exploited vulnerability in web applications. Existing XSS testing tools utilize a brute-force or heuristic approach to discover vulnerabilities, which increases the testing time and load of the target system. Reinforcement learning (RL) is expected to decrease the burden on humans and enhance the efficiency of the testing task. This paper proposes a method to automate XSS vulnerability testing using RL. RL is employed to obtain an efficient policy to compose test strings for XSS vulnerabilities. Based on an observed state, an agent composes a test string that exploits an XSS vulnerability and passes the string to a target web page. A training environment *XSS Gym* is developed to provide a variety of XSS vulnerabilities during training. The proposed method significantly decreases the number of requests to the target web page during the testing process by acquiring an efficient policy with RL. Experimental results demonstrate that the proposed method effectively discovers XSS vulnerabilities with the fewest requests compared to the existing open-source tools.

1 INTRODUCTION

Since recent computer systems have become more complicated, security protection is a growing concern. The cyber-space attacks are also becoming more sophisticated, and the defense of computer systems must be enhanced. From the defender's viewpoint, reinforcement learning (RL) is expected to provide the opportunity for prior vulnerability testing (Song and Alves-Foss, 2015; Avgerinos et al., 2018). Thus, applying RL to cybersecurity, such as autonomous attacks and vulnerability detection, has emerged as a key research topic in recent years (Meyer et al., 2021; Nguyen and Reddi, 2021).

In this paper, we focus on vulnerability testing in network-attached devices. A well-known vulnerability is *cross-site scripting (XSS)*, which is recognized as one of the most frequent threats (OWASP Top 10 team, 2021). XSS vulnerabilities allow attackers to execute malicious scripts on the web application of unsuspecting users by improperly handling external input strings. Existing XSS vulnerability testing tools utilize brute-force or heuristic methods based on known attack patterns, which increases the number of requests to the target web page, thus increasing the testing time and load on the web server. Considering the increase in network-attached devices, such as IoT devices with limited computational resources, an efficient vulnerability testing method must be established.

This paper proposes a method to automate XSS vulnerability testing using RL to understand the nature of autonomous attacks. Here, an autonomous attack represents that a policy in an RL agent is trained using a training environment to select an efficient attacking action that is adapted to a target environment. The proposed method composes a *test string* that exploits a vulnerability by combining known attack string fragments used in XSS attacks and state observation by parsing the source code of the target web page. RL is employed to obtain an efficient policy to autonomously compose the test string without human intervention. The experimental results demonstrate that the proposed method can discover vulnerabilities with the fewest requests compared to existing open-source tools.

The contributions of the paper can be summarized as follows:

- We define state observations by parsing the source code of a target web page, agent actions by the string combination, and reward by the current state observation to implement an XSS vulnerability testing method using RL.
- Based on the state, action, and reward, we propose

[a] https://orcid.org/0000-0002-6517-1703
[b] https://orcid.org/0000-0003-2571-0116

Hasegawa, K., Hidano, S. and Fukushima, K.
Automating XSS Vulnerability Testing Using Reinforcement Learning.
DOI: 10.5220/0011653600003405
In *Proceedings of the 9th International Conference on Information Systems Security and Privacy (ICISSP 2023)*, pages 70-80
ISBN: 978-989-758-624-8; ISSN: 2184-4356
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

an XSS vulnerability testing method using RL.

- We develop a training environment called *XSS Gym* that randomly provides vulnerable web pages based on pre-defined templates and parameters. XSS Gym facilitates the RL agent to experience a large number of XSS vulnerability patterns compared to manually setting up static vulnerable pages.
- We experimentally demonstrate that the proposed method discovers XSS vulnerabilities with the fewest requests compared to existing open-source tools.

2 BACKGROUND

This section presents the background of RL and XSS vulnerabilities.

2.1 Reinforcement Learning (RL)

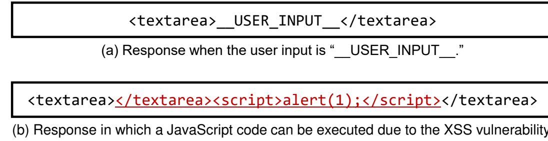
RL represents an algorithm aimed at obtaining the optimal policy by maximizing the expected reward passed from the environment. The environment is often modeled as a Markov decision process (MDP), which is defined using the following elements:

- state space $\mathcal{S}$ that the environment can take;
- action space $\mathcal{A}$, including all actions that the agent can perform;
- state transition function $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$, which is the probability of transition to state s_{t+1} when action a_t is performed in state s_t at time t;
- immediate reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$;
- reward discount factor $\gamma \in [0,1]$.

$\pi(a \mid s)$ is the policy in which action a can be performed in state s. The goal of RL is to maximize the expected discount cumulative reward under policy π through the interaction of the environment and agent.

2.2 Cross-Site Scripting (XSS)

XSS is a vulnerability in a web application in which a malicious script can be injected into the web application, and the vulnerability allows attackers to execute the malicious script on the web application. An attacker exploits the XSS vulnerabilities to execute an arbitrary script, *payload*, on the system of the web application users and may steal confidential information or perform malicious actions unintentionally.



(a) Response when the user input is "__USER_INPUT__."

(b) Response in which a JavaScript code can be executed due to the XSS vulnerability.

Figure 1: Example of an XSS attack.

Figure 1 shows an example of the XSS attack. Figure 1 (a) shows the response of a web application when the user input is '`__USER_INPUT__`.' Figure 1 (b) shows the attack case: a malicious attacker inputs the underlined red string, which is a JavaScript code with a closing tag of `textarea`. The closing tag of `textarea` at the beginning of the input string closes the `textarea` context, and the next `script` tag is valid as an HTML code. Thus, the JavaScript code, '`alert(1);`' is executed in the web application. Such an input string to a web application can be passed as the parameter of a GET request or payload of a POST request. Suppose an attacker embeds a malicious code into a URL as a GET parameter and distributes the URL. In this case, a user who accesses the URL will execute the malicious code injected into the web application.

XSS is classified into three types: reflected, stored, and DOM-based. In reflected XSS, a part of an input string is directly reflected on a web server output. In stored XSS, the input string is provided to the web application and stored, for instance, in a database. DOM-based XSS is the case in which the input string is directly reflected on the content without being passed through the web server. The nature of the XSS vulnerability is the same across the three types. This paper focuses on reflected and DOM-based XSS attacks because the test strings are immediately reflected on target applications.

Although XSS attacks can be prevented by a sanitization process for the input queries, vulnerability may remain due to the lack of security awareness or potential bugs in external software libraries. A web application firewall (WAF) can protect a web application from several XSS attacks. However, WAF protection cannot address DOM-based attacks, and the processing of WAF service may be heavy for resource-limited devices. Therefore, vulnerability testing before deployment is required.

2.3 Autonomous Attack Using RL

Existing XSS vulnerability testing tools often adopt a brute-force or heuristic approach to check whether an attack string exploits the vulnerability. The brute-force approach can exhaustively examine the vulnerability, but the load on the web server and testing time may increase. Although the impact on the server load

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

may be negligible in the case of recently developed high-performance web servers, the load is significant for IoT devices with web interfaces to configure the device and monitor the sensors. Such web interfaces may be vulnerable to XSS attacks because of the low cost and low performance of devices. In this situation, vulnerability testing with numerous requests is impractical. Thus, it is necessary to decrease the load on the devices and testing time.

Autonomous attack methods have been recently studied (Zennaro and Erdodi, 2020; Erdödi and Zennaro, 2022; Erdödi et al., 2021; Caturano et al., 2021; Demetrio et al., 2020). These methods try to access hidden files that can be recognized from the URLs specific to open-source or popular web applications, exploit vulnerabilities of the target system, or launch SQL injection attacks. Furthermore, RL can be applied for penetration testing on network systems (Hu et al., 2020; Bland et al., 2020; Chowdary et al., 2020; Ghanem and Chen, 2020).

In (Frempong et al., 2021), an automated exploit generation method for a JavaScript XSS vulnerability, called HIJaX, is proposed. Although HIJaX can generate various XSS attack codes, the algorithm does not consider filter evasion that is adapted to the web page being inspected.

In (Caturano et al., 2021), a reflected-XSS attack method by crafting attack strings using RL is proposed. This method divides an attack string into five sections, and a list of attack string fragments is composed based on known attack strings. The Q-learning algorithm obtains the policy to compose the appropriate attack string for the target web application by combining the attack string fragments in the five sections. In (Caturano et al., 2021), the number of requests to detect reflected-XSS vulnerabilities is significantly smaller than that in the existing open-source tools for XSS testing. However, human interaction is required to observe the state during training because the method is based on the human-in-the-loop technique. Therefore, a person with expert knowledge of XSS is necessary. These problems must be solved to realize completely autonomous testing.

Our Goal: Our goal is to automate XSS vulnerability testing with a few attempts such that defenders can test their web applications efficiently without expert knowledge. RL can be used to compose test strings autonomously and efficiently. However, the settings for an RL agent and preparation of the training environment are the problems. In this paper, we propose a method to automate XSS vulnerability testing and a training environment.

3 PROPOSED METHOD

This section presents a method to automate XSS vulnerability testing using RL.

3.1 Overview

We establish an XSS vulnerability testing method using RL. The proposed method uses RL to obtain an efficient policy to compose test strings autonomously through string-combining operations and state observations based on the parsing of web pages.

Attackers must add strings before and after the payload such that the payload is reflected in a web application content as an executable script. Then, the payload becomes executable, and the XSS attack succeeds. We define the complete string obtained through such an operation as the *test string*.

As mentioned in Section 2.1, the RL algorithm can obtain the optimal policy π in the environment that follows a Markov decision process or can approximate it. Therefore, it is necessary to determine the action $\mathcal{A}$, state $\mathcal{S}$, and reward r to obtain an efficient policy. This paper defines these items as follows (details are presented in the following sections):

- $a \in \mathcal{A}$: an operation of composing a test string.
- $s \in \mathcal{S}$: a parser state for the payload string.
- r: determined based on the number of steps required to achieve the goal.

Figure 2 shows an overview of the proposed method. The action selection, state observation, and reward acquisition are repeatedly performed between the agent and environment (Section 3.2, Section 3.3, and Section 3.4). The agent implements RL and involves the policy that determines the next action based on the current state (Section 3.5). The environment is the target web application to be tested. A training environment, called XSS Gym, is proposed in Section 3.6. The proposed method aims to acquire an efficient policy to compose the test string that successfully exploits XSS vulnerability.

3.2 Action

The key task in the proposed method is the composition of the test string by adding strings before and after the payload script. First, the test string is split into four sections. Next, the operations on the sections are defined as actions to compose the test string.

3.2.1 Sections of a Test String

The test string is split into four sections to simplify the composition of a test string.

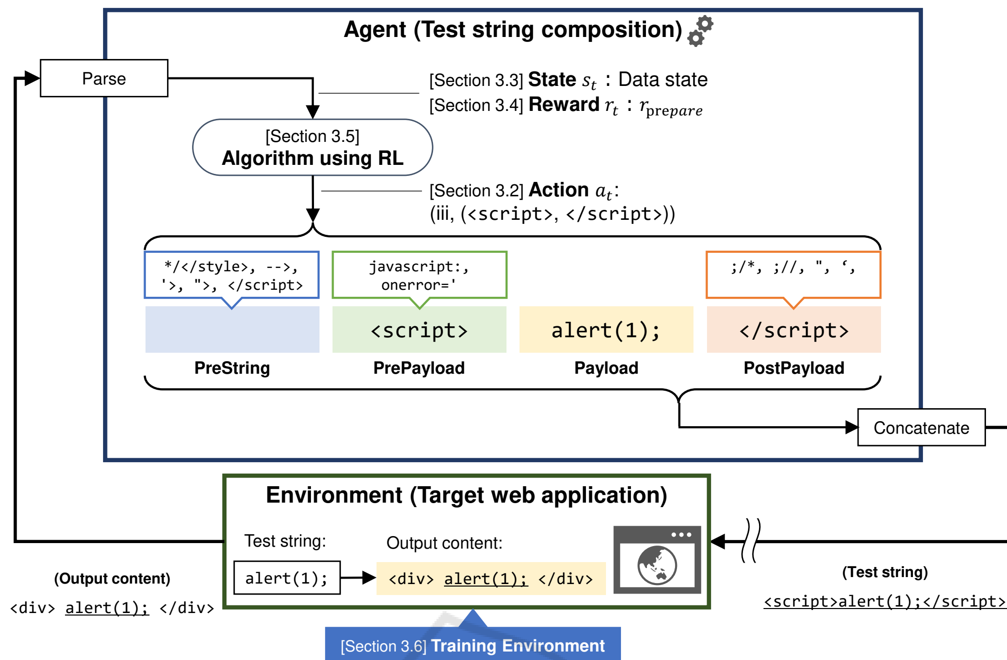


Figure 2: Overview of the proposed method.

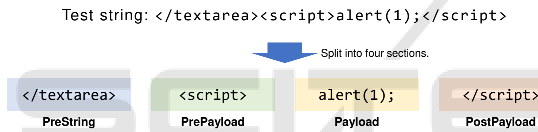


Figure 3: Sections of a test string.

1. **PreString:** The string in this section closes the previous context.
2. **PrePayload:** The string in this section starts the new context for rendering the payload executable.
3. **Payload:** The string in this section is an arbitrary script to be executed on the web application.
4. **PostPayload:** The string in this section closes the context of the payload.

Figure 3 shows an example of a test string and its sections. In the test string, 'alert(1);' is the payload script, which belongs to the Payload section. The script tag encloses the payload script. According to the section definitions, the starting tag '<script>' belongs to the PrePayload section, and the closing tag '</script>' belongs to the PostPayload section. The first piece of the test string '</textarea>' closes the textarea context that is originally displayed by the web application and belongs to the PreString section.

3.2.2 Components of an Action

The operations on the four sections are defined as actions to compose the test string. An action $a \in \mathcal{A}$ is defined as the tuple (*target*, *content*). The *target* element shows the target section to be operated on. The *content* element shows the content of the operation. The remaining section describes the target and content elements.

Target of an Action. In terms of the target of an action, this paper defines five targets based on the four sections introduced above. Because an attacker arbitrarily determines the script of the Payload section, our algorithm does not change this section. Other sections, a pair of sections, and the whole string are the targets for the actions.

1. **Target 1: PreString:** The action targeting this section closes the previous context and changes the context of the following sections.
2. **Target 2: PrePayload:** The action targeting this section changes the current context to the new context for the Payload section.
3. **Target 3: PrePayload and PostPayload:** The action targeting these sections encloses the Payload section with a specified tag and can change to a different context with only one operation.
4. **Target 4: PostPayload:** The action targeting this section closes the context of the Payload section.
5. **Target 5: Whole String:** This action converts (e.g., encodes) the whole string.

Targets 1, 2, and 4 focus on the PreString, PrePayload, and PostPayload sections, respectively. Tar-

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

get 3 simultaneously focuses on the PrePayload and PostPayload sections. It is useful to consider that a pair of sections are simultaneously changed because the PrePayload and PostPayload sections often correlated according to the known test strings for XSS vulnerabilities. For example, when we wish to enclose the Payload section with the `script` tag, it is necessary to set ‘<script>’ for the PrePayload section and ‘</script>’ for the PostPayload section. Target 5 corresponds to the conversion of the whole text, such as changing the text encoding to another one. The text encoding can be changed to fake the XSS detector and is effective in evading the pattern-matching mechanism.

Content of an Action. In terms of the content of action, a test string can be composed by replacing the string in the specified section(s) with another string. The string in the target section defined in the previous section is replaced with another string. To prepare the strings to be placed in each section, the string fragments are collected from the known test strings, and the string fragment list is generated. The string fragment list stores string fragments and their corresponding targets. The string fragments include null strings for each section to remove the string in a target.

Another operation is converting the string in the target section(s) to the specified encoding method. For example, the UTF-7 encoding expresses the character ‘<’ and ‘>’ as ‘`+ADw-`’ and ‘`+AD4-`’, respectively. Thus, the starting tag of the script context ‘<script>’ is converted to ‘`+AD-script+AD4-`.’ Therefore, pattern matching protection can be evaded by this conversion. Although this exploit does not work for modern web browsers with an ordinal situation, several old systems might still be vulnerable.

3.2.3 Action Space

An action in the action space is represented by the tuple of the *target* and *content*, as discussed. The action space is constructed before the training based on a training dataset.

3.3 State

The state definition is of significance to efficiently obtain an optimal policy by RL. It is desirable to be able to mechanically represent the state of the source code of a web page. This paper introduces the parsing of the source code to represent states. Furthermore, the states are defined to efficiently estimate the reward discussed later.

3.3.1 State Definition Based on Parsing

In the proposed method, the source code obtained as a response from the target web application is parsed. Through parsing, the state of the payload script (the string in the Payload section) is observed for RL.

Table 1 shows examples of the parsing states for the payload script, ‘`alert(1);`.’ The second column lists the responses from a web server, and the third column lists the states of the payload script (the fourth column is introduced later). In this table, we refer to the HTML5 specification (WHATWG,) to recognize the state.

In row (a), the user’s input is reflected inside the `div` tag in the response. The `div` tags are often used to divide the sections of the contents on the web page. According to the HTML5 specification, the string directly inside the `div` tag is identified as ‘Data state.’ The string at ‘Data state’ is displayed as text and is not executable.

In row (b), the user’s input is reflected inside the `textarea` tag in the response. The `textarea` tags are used to provide an input box that accepts multiline strings from users. The string directly inside the `textarea` tag is identified as the ‘RCDATA state’ according to the HTML5 specification. The string at the ‘RCDATA state’ is displayed as text and is not executable. In contrast to those in the ‘Data state,’ the ‘RCDATA state’ strings are no longer parsed as an HTML code until the end of the ‘RCDATA state,’ whereas HTML tags inside the ‘Data state’ strings are recognized. Therefore, even if the user’s input contains the `script` tag with a payload script, it is displayed inside the `textarea` tag.

In row (c), the user’s input is reflected inside the `script` tag in the response. The string directly inside the `script` tag is identified as the ‘Script data state.’ The string at the ‘Script data state’ is executable as a script on the web application. Therefore, if a malicious script is included in the user’s input and identified as ‘Script data state,’ it is unintentionally executed by the user.

As described, the parser state can help identify whether the payload script is executable or not. In the proposed method, the parser state is observed and used as a state for RL. A simple method to realize autonomous XSS vulnerability testing is to generate a test string that actually exploits an XSS vulnerability. Thus, we aim to ensure that the payload string is executable as a script.

3.3.2 State Sets

To systematically consider the parser states, they are classified into sets based on the steps to the state at

Table 1: Examples of states.

	Response	State of alert(1);	State set	Executable?	Reward
(a)	alert(1);	Data state	$\mathcal{S}_1$		r_{prepare}
(b)	<textarea>alert(1);</textarea>	RCDATA state	$\mathcal{S}_2$		r_{other}
(c)	<script>alert(1);</script>	Script data state	$\mathcal{S}_0 = \mathcal{S}_g$	✓	r_{goal}

which the payload script is executable.

As mentioned, our goal is to ensure that the payload string is in the state in which the string is executable. This state, known as the *goal state*, is defined as follows:

Definition 1 (Goal State). *$\mathcal{S}_{\mathrm{g}}$ is a set of goal states that are parser states in which a string is executable as a script on the web application.*

Hereafter, our goal is to ensure that the payload string is in a set of goal states $\mathcal{S}_{\mathrm{g}}$.

Next, the parser states other than the goal states are classified. When the proposed method composes a test string, it is helpful to estimate how many steps are needed to achieve one of the goal states.

Definition 2 (Distance to the Goal State). *$\mathcal{S}_d$ is a set of states whose steps to any goal state are d. Here, $\mathcal{S}_0 = \mathcal{S}_{\mathrm{g}}$ and $s_0 \in \mathcal{S}_0$. $s_d \in \mathcal{S}_d$ is recursively defined as follows:*

- *s is a state, and goal state $s_0 \in \mathcal{S}_0$ is reached after action a is applied from s. If $s \notin \mathcal{S}_0$, the number of steps to any goal state d is 1, and s is an element of $\mathcal{S}_1$, $s_1(s_d, d = 1)$.*
- *s is a state, and state $s_n \in \mathcal{S}_n$ is reached after action a is applied from s. If $s \notin \mathcal{S}_i, 0 \le i \le n$, the number of steps to any goal state d is $n+1$, and s is an element of $\mathcal{S}_{n+1}$, $s_{n+1}(s_d, d = n+1)$.*

The fourth column in Table 1 shows the steps to any goal state. The 'Data state' in row (a) is categorized as $\mathcal{S}_1$ because enclosing the payload script with a `script` tag renders the script executable. The 'RCDATA state' in row (b) is categorized as $\mathcal{S}_2$. In the 'RCDATA state,' even if the payload script is enclosed with a `script` tag, the script does not become executable. The `textarea` context must be closed to render the script executable. Since the operation involves two steps, the 'RCDATA state' is categorized as $\mathcal{S}_2$. The 'Script data state' in row (c) is categorized as $\mathcal{S}_0$ (i.e., $\mathcal{S}_{\mathrm{g}}$) because the script has already been executable, as shown in the fifth column.

3.4 Reward

As described in the previous section, the state sets are defined based on the steps to the *goal state*. It is helpful to estimate the step to the *goal state*. In this section, the reward for RL is defined based on the state sets.

The reward types are defined as follows:

- r_{goal}: reward when the environment achieves any goal state $s_{\mathrm{g}} \in \mathcal{S}_{\mathrm{g}}$.
- r_{prepare}: reward when the environment achieves state $s_1 \in \mathcal{S}_1$ where the minimum number of steps to any goal state is 1.
- r_{other}: reward when the environment achieves state $s_i \in \mathcal{S}_i, i \ge 2$ where at least two steps are required to achieve any goal state.

Real values are assigned to the rewards. The relationship between the rewards is defined as follows:

$$r_{\mathsf{goal}} > r_{\mathsf{prepare}} > r_{\mathsf{other}} \quad (1)$$

The reward values used in the experiment are shown in Section 4.

The simplest way to determine the reward is to assign a reward only if an attack actually exploits a vulnerability. However, many test strings must be considered until any XSS vulnerability is exploited. Therefore, this paper considers the state sets defined in the previous section to determine the reward efficiently. Table 1 lists the examples of the relationship between the state and reward. Row (a) is the 'Data state' that is categorized as $\mathcal{S}_1$. This state requires only one step to a goal state. As shown in the sixth column, the reward becomes r_{prepare}, according to the definition. Similarly, the rewards of rows (b) and (c) are determined as r_{other} and r_{goal}, respectively.

3.5 Agent

An agent can be enhanced to search for an optimal policy by applying the two mechanisms.

One mechanism is an LSTM-based policy. As mentioned in Section 2.1, an environment is often modeled as MDP for RL. Although the observation states are carefully set up, completely observing the internal state of an environment is difficult. A partially observable MDP (POMDP), in which an agent can observe a part of an actual internal state, can be employed in security. In a POMDP, an observation implies several states probabilistically. Since the probabilities can be estimated based on the trajectory

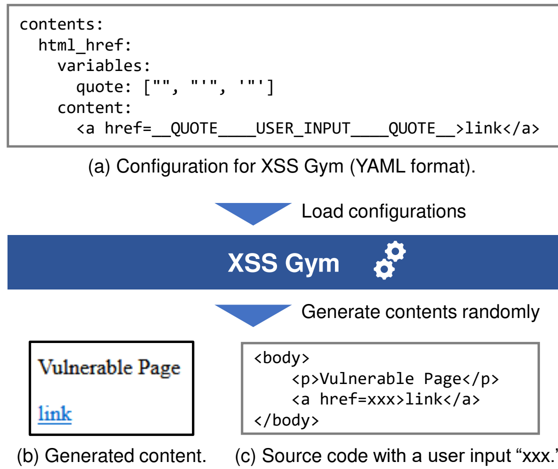


Figure 4: Example of XSS Gym.

of observations, an LSTM-based policy is used to predict the current internal state.

The second mechanism is an intrinsic curiosity module (ICM) (Pathak et al., 2017). The module makes the agent explore unpredicted responses from the environment. We compare the training results with or without the ICM in Section 4.

3.6 Training Environment: *XSS Gym*

We propose a training environment for XSS vulnerability testing, called *XSS Gym*, to effectively learn various XSS vulnerabilities.

The training environment must behave in a manner that mimics real-world web applications. However, the training web applications for security beginners provide only a limited number of vulnerable web pages. To solve the problem, XSS Gym provides various web pages that are randomly configured based on several templates and parameters. First, a set $\mathcal{T}$ of web page templates in which a given string is shown as a content is prepared. Each template $\tau \in \mathcal{T}$ has several parameters $\mathcal{P}_\tau$. Figure 4 shows an example of XSS Gym. Figure 4 (a) shows a configuration for XSS Gym that is described in a YAML format. In the configuration, 'content' shows a template τ, and 'variables' shows a list of parameters $\mathcal{P}_\tau$ for the template τ. In the example, the string `__QUOTE__` in the template is randomly replaced with either (no character), `'`, or `"`. Then, a content is generated as shown in Figures 4 (b) and (c), in which `__USER_INPUT__` in the template is replaced with an user input '`xxx`.'

During the training of RL, XSS Gym continuously provides web pages. At the beginning of an episode, XSS Gym randomly chooses a template $\tau \in \mathcal{T}$ and configures the web page with parameters $\rho \in \mathcal{P}_\tau$. The template and parameters are not changed during the episode. The agent sends a signal to XSS Gym at the beginning and end of the episode for cooperation.

To mimic the real-world situations in which an XSS sanitization is partially applied, a set $\mathcal{F}$ of filter configurations is also prepared. The web page behaves differently by randomly applying several variations of XSS filters, even if the content looks the same.

3.7 Vulnerability Testing

The RL model is trained based on the action, state, and reward. However, the introduced model does not completely follow the Markov model. Specifically, the transition function P is stochastic and not deterministic because several web applications often filter out test strings. This aspect must be considered to establish a vulnerability testing algorithm using RL.

Algorithm 1: XSS vulnerability testing.

Input: Trained model $\mathcal{M}$, Environment $\mathcal{E}$, Payload string X
Output: Test string T

1: $T \leftarrow X$, $\mathcal{H} \leftarrow \emptyset$, $s \leftarrow$ Initial state
2: **while** $s \notin \mathcal{S}_g$ **do**
3: $L \leftarrow \{(a,p) \mid p = \pi(a \mid s), a \in \mathcal{A}\}$ // Obtain next actions and their probabilities from model $\mathcal{M}$.
4: Sort L with respect to p in descending order.
5: $i \leftarrow 0$, $(a,p) \leftarrow L[i]$
6: **while** $(s,a) \in \mathcal{H}$ and $i < |L|$ **do**
7: $i \leftarrow i+1$
8: $(a,p) \leftarrow L[i]$
9: **end while**
10: **if** $i == |L|$ **then**
11: **return** null // Not found
12: **end if**
13: $\mathcal{H} \leftarrow \mathcal{H} \cup \{(s,a)\}$
14: Perform action a and update T.
15: $s \leftarrow$ Observe a state from environment $\mathcal{E}$.
16: **end while**
17: **return** T

Before vulnerability testing, the agent learns the training dataset and obtains a policy. Training can be performed through the normal RL process. In the training phase, an agent performs an action according to the current policy and composes a test string. The test string is provided to the target web application, and the web application returns the response. The agent observes the state of the payload string and obtains the reward according to the state. This process is repeated multiple times for various target web pages. Finally, a model that obtains an efficient policy

for composing a test string is established.

Algorithm 1 describes the process flow for XSS vulnerability testing. This algorithm repeatedly composes a test string and attempts to exploit the vulnerability using the string. If an exploit is successful, the algorithm returns the successful test string. If the algorithm cannot find the appropriate test string within a specified number of iterations, the algorithm returns null and notifies the user that no XSS vulnerabilities are discovered.

4 EVALUATION

This section describes the evaluation of the proposed method using a vulnerable web application. As mentioned in Section 2, our goal is to automate XSS vulnerability testing. The proposed method uses RL to compose test strings autonomously and efficiently. The experiments aim to answer the following research questions:

RQ1: Does XSS Gym provides appropriate samples for training an RL agent?

RQ2: Does the agent obtain an efficient policy to compose a test string?

4.1 Setup

The programs are implemented in Python. PPO (Schulman et al., 2017) is applied as an RL algorithm. In the experiments, we use the Ray[1] library to implement a PPO algorithm.

Training. We train RL agent using *XSS Gym*. The template and parameters are prepared based on the existing vulnerable web pages in WAVSEP (Chen, 2014) and Webseclab (Yahoo Inc., 2020). The program and vulnerability testing tools run as Docker containers and are connected via a virtual network.

We prepare four settings. Random uses XSS Gym as a training environment and an LSTM network as a policy network in RL. Weighted uses a weighted version of XSS Gym, in which XSS Gym gives priority to providing the templates that have not yet been selected or for which an agent takes a large number of requests in previous episodes. Random+ICM (resp. Weighted+ICM) is configured based on Random (resp. Weighted), but an ICM is employed when exploration.

The model is trained for 100k steps with a batch size of 1000 steps. The learning rate is 0.001, and the GAE parameter in (Schulman et al., 2017) is 0.95. Other parameters are set to the default settings of Ray.

For the reward values, we assign a positive value to r_{goal} and 0 to r_{prepare}. r_{other} is set to a negative value that is decreased as the number of steps in an episode increases.

Testing. Algorithm 1 is performed using the trained model. Since the agent produces the same test string several times in certain cases, we count the number of unique requests during the evaluation. The open-source tools that scan XSS vulnerabilities in web pages are used in the experiments to confirm that the proposed method can obtain an efficient policy. We select the tools that are available online for free and maintained in 2020 or later. Table 2 lists the tools used in the experiments. We count the number of requests until a vulnerability is detected.

We use 19 web pages in Webseclab (Yahoo Inc., 2020), which contains several web pages vulnerable to XSS, as target web pages.

Table 2: Open-source tools used in the experiments.

Tool	URL
XSpear	https://github.com/hahwul/XSpear
XSSer	https://github.com/epsylon/xsser
XSSMap	https://github.com/Jewel591/xssmap
Wapiti	https://github.com/wapiti-scanner/wapiti
w3af	https://github.com/andresriancho/w3af

4.2 Results

4.2.1 Training Using XSS Gym

To answer RQ1, we evaluate XSS Gym. We trained the agent using XSS Gym. Figure 5 shows the success rate of exploiting an XSS vulnerability on a target web page. The x-axis shows the episode number, and the y-axis shows the success rate during the last 500 episodes. In Figure 5, all plots show a trend of increasing success rate. In particular, Random and Random+ICM slightly outperform the Weighted (with and without ICM) settings. This is because the agents with the weighted version of a training environment train intuitively difficult web pages to exploit XSS vulnerabilities due to the weighting mechanism of XSS Gym. In any case, the agent could improve the success rates as the number of episodes increases. From the results, XSS Gym provides appropriate vulnerable web pages for training an RL agent.

4.2.2 Evaluation on Vulnerable Web Pages

To answer RQ2, we evaluate the proposed method using the Webseclab pages. The Webseclab pages, all

[1]https://github.com/ray-project/ray

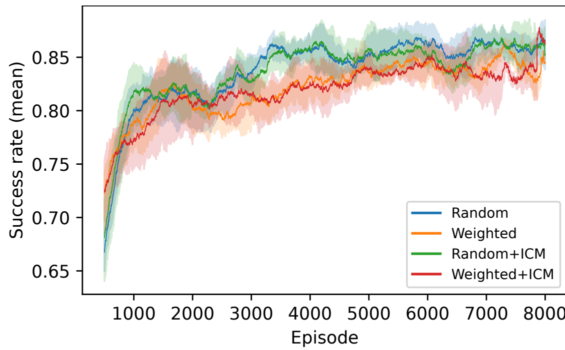


Figure 5: Success rate when training.

Table 3: Average results of each setting (five trials).

Setting	Requests	(s.d.)	Success rate
Random	160.0	(11.4)	0.853
Weighted	149.2	(25.1)	0.874
Random+ICM	175.2	(58.5)	0.863
Weighted+ICM	**122.4**	(16.7)	**0.884**

of which have XSS vulnerabilities, are tested with the trained agent.

We evaluated the agents with four settings in Section 4.1. We tried five trials with different random seed numbers and obtained the averaged results. Table 3 shows the averaged number of requests and success rates to exploit the web pages in Webseclab. As shown in Table 3, Weighted+ICM shows the fewest requests and obtained the largest success rate in exploiting XSS vulnerabilities in the tested web page.

We focus on the agent setting with the best result in the previous evaluation. Table 4 shows the target pages accessed in the experiment and the unique number of requests with the Weighted+ICM setting. As shown in Table 4, the agent exploits XSS vulnerabilities in each page within 33 requests. The proposed method requires less than ten requests for 16 pages to detect the XSS vulnerability. The trained agent successfully obtains an efficient policy and selects attacking actions that are adapted to a target web page.

Figure 6 and Figure 7 show the comparison results of the proposed method with the open-source tools. Figure 6 shows the total number of requests to complete the testing process. Since the open-source tools aim to cover all XSS vulnerabilities in the testing, a number of test strings are queried. Although this strategy covers many XSS vulnerabilities, several test strings are inapplicable to the target content. In contrast, the proposed method requests the most suitable test string considering the state of the content. Therefore, the total number of requests is considerably smaller than the other tools.

We count the minimum requests to detect at least one XSS vulnerability by the open-source tools. Here,

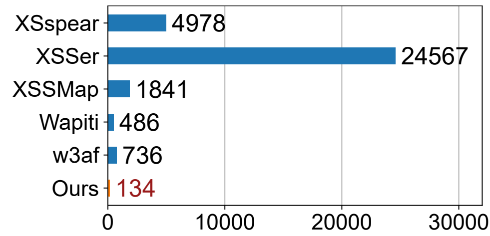


Figure 6: Total number of requests.

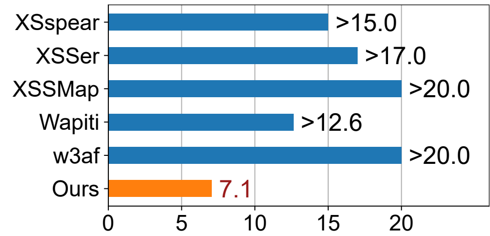


Figure 7: Average minimum number of requests.

we average the count over the 19 web pages. The number is considered to be 20 when more than 20 requests are required to exploit an XSS vulnerability in existing tools. Figure 7 shows the results of the average minimum requests. The sign '>' shows that one or more cases require more than 20 requests. Therefore, the actual average counts are larger than the shown counts. As shown in Figure 7, the proposed method shows the fewest requests among the open-source tools. This finding implies that the proposed method obtains an efficient policy during the training phase and thus, successfully detects an XSS vulnerability with the fewest requests compared to the recent open-source tools.

4.3 Limitation

Since the agent composes the test string based on the pre-defined action space, it cannot address unknown XSS vulnerabilities. Introducing a natural language processing technique, which is applied in (Frempong et al., 2021), and using a generative adversarial network would be solutions to address the problem. However, how to efficiently integrate such techniques must be considered.

The quality of the obtained policy depends on the training dataset. Typically, RL can learn experienced actions and corresponding rewards. Therefore, web pages that involve various XSS vulnerabilities are needed. XSS Gym partially solves the problem by randomly generating vulnerable web pages based on given templates and parameters. However, how to prepare the templates and parameters is remained to be considered. Collecting real-world web application logs and analyzing XSS exploitations from them can

Table 4: Number of requests pertaining to the proposed method.

Page	Requests	Page	Requests
backslash1	3	js6_sq_combo1	4
basic	2	js_script_close	13
basic_in_tag	2	oneclick1	24
doubq1	6	onmouseover	9
enc2	33	onmouseover_div_unquoted	6
full1	2	onmouseover_unquoted	8
js3	1	rs1	2
js3_notags	1	textarea1	4
js4_dq	6	textarea2	4
js6_sq	4		

be a solution. More work is still needed to enhance training environments.

5 CONCLUSION

This paper presents an XSS vulnerability testing method using RL and a training environment, XSS Gym. The proposed method trains an RL agent to autonomously compose test strings by replacing the fragments of known test strings and observing the parsing of the target web page. Since RL obtains an efficient policy for composing test strings, the number of requests for testing web pages is drastically decreased. The experimental results demonstrate that an RL agent can be trained using XSS Gym and the proposed method discovers vulnerabilities in web pages with the fewest requests compared to other existing vulnerability testing tools.

REFERENCES

Avgerinos, T., Brumley, D., Davis, J., Goulden, R., Nighswander, T., Rebert, A., and Williamson, N. (2018). The mayhem cyber reasoning system. *IEEE Security & Privacy*, 16(2):52–60.

Bland, J. A., Petty, M. D., Whitaker, T. S., Maxwell, K. P., and Cantrell, W. A. (2020). Machine learning cyberattack and defense strategies. *Comput. Secur.*, 92:101738.

Caturano, F., Perrone, G., and Romano, S. P. (2021). Discovering reflected cross-site scripting vulnerabilities using a multiobjective reinforcement learning environment. *Computers & Security*, 103(102204).

Chen, S. (2014). WAVSEP - the web application vulnerability scanner evaluation project.

Chowdary, A., Huang, D., Mahendran, J. S., Romo, D., Deng, Y., and Sabur, A. (2020). Autonomous security analysis and penetration testing. In *Proc. International Conference on Mobility, Sensing and Networking*.

Demetrio, L., Valenza, A., Costa, G., and Lagorio, G. (2020). Waf-a-mole: Evading web application firewalls through adversarial machine learning. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1745–1752. Association for Computing Machinery.

Erdödi, L., Åvald Åslaugson Sommervoll, and Zennaro, F. M. (2021). Simulating sql injection vulnerability exploitation using q-learning reinforcement learning agents. *Journal of Information Security and Applications*, 61:102903.

Erdödi, L. and Zennaro, F. M. (2022). The agent web model: modeling web hacking for reinforcement learning. *International Journal of Information Security*, 21(2):293–309.

Frempong, Y., Snyder, Y., Al-Hossami, E., Sridhar, M., and Shaikh, S. (2021). HIJaX: Human intent javascript xss generator. In *SECRYPT*, pages 798–805.

Ghanem, M. C. and Chen, T. M. (2020). Reinforcement learning for efficient network penetration testing. *Information*, 11(1):6.

Hu, Z., Beuran, R., and Tan, Y. (2020). Automated penetration testing using deep reinforcement learning. In *Proc. EuroS&P Workshops*, pages 2–10.

Meyer, T., Kaloudi, N., and Li, J. (2021). A systematic literature review on malicious use of reinforcement learning. In *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, pages 21–28.

Nguyen, T. T. and Reddi, V. J. (2021). Deep reinforcement learning for cyber security. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–17.

OWASP Top 10 team (2021). *OWASP Top 10:2021*. https://owasp.org/Top10/.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning*, pages 2778–2787.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

Song, J. and Alves-Foss, J. (2015). The darpa cyber grand challenge: A competitor's perspective. *IEEE Security & Privacy*, 13:72–76.

WHATWG. *HTML Standard*. https://html.spec.whatwg.org/.

Yahoo Inc. (2020). *Webseclab*. https://github.com/yahoo/webseclab.

Zennaro, F. M. and Erdodi, L. (2020). Modeling penetration testing with reinforcement learning using capture-the-flag challenges and tabular q-learning.