機械学習による全自動 Web 脆弱性スキャナの試み

Experiment of full automatic web vulnerability scanner using Machine Learning

三井物産セキュアディレクション株式会社　高江洲 勲

Abstract: 著者は Web アプリケーションの脆弱性を全自動で検出する脆弱性スキャナ「SAIVS」の開発を進めており、人間のセキュリティエンジニアと同等のレベルで脆弱性診断を行うことを最終目標としている。現時点の SAIVS はベータ版であるが、シンプルな Web アプリケーションにおいて、人間と同じように Web ページをクローリングすることが可能である。また、Web アプリケーションの挙動を観察し、脆弱性を検出するための最適な検査値を自ら考案し、少ない手数で効率よく脆弱性を検出することが可能である。この人間のような行動を、複数の機械学習アルゴリズムを組み合わせることで実現している。本稿では、SAIVS を実現するための手法と検証実験の結果を報告する。

Keywords: Vulnerability Assessment, Full automatic, Machine Learning, Web Application, Vulnerability Scanner

1. 研究の背景

日本におけるサイバー攻撃の件数は年々増加の一途を辿っており、攻撃手法も巧妙化し、脅威は増大している[1]。また、2020 年の東京オリンピック・パラリンピックに向けて、関連組織やシステムを狙った攻撃も今後増加していくものと思われる。これらの攻撃に対応するためには、高度なスキルを持ったセキュリティ人材の確保が必要であるが、現状では情報セキュリティに係る人材は大幅に不足している。情報処理推進機構の試算によると、特に製造業や卸売業、医療及び福祉等のユーザ業種において顕著になっており、現在情報セキュリティに従事するエンジニア 26.5 万人の内、約 16 万人はスキルが不足していると言われている。この人材不足を解決するために、セキュリティ・キャンプや SECCON 等の CTF 大会を通して人材育成及びスキルの底上げが図られているが[2]、今後より一層の労働力不足が予測される日本の労働市場において[3]、十数万人規模の高度なスキルを持った人材を確保することは困難であると思われる。

一方、米国では人材不足を解決するために、人工知能（以下、AI）を情報セキュリティに活用する動きが活発化している。2016 年 8 月には、ラスベガスにおいて DARPA 主催の AI による世界初のコンピュータハッキング大会「Cyber Grand Challenge」が開催された[4]。本大会は、相手チームから攻撃を受ける前に自チームのソフトウエアに存在するゼロデイ脆弱性にパッチを当てつつ、相手チームのソフトウエアの脆弱性を利用してサーバに侵入し、フラグを奪う競技であり、この一連の行為を全自動で行うものである。本大会が開催された背景として、現在は脆弱性の検出やパッチ当てを熟練したエンジニアに頼っており、迅速かつ広範囲な脆弱性対応が難しいという現状がある。なお、製品のゼロデイ脆弱性が悪用されてから製品ベンダが脆弱性に気付くまでに平均 312 日かかると言われている[5]。本大会は、数分以内に脆弱性を自動で対処できるようにすることで、人材不足という大きな課題を解決することを目的としている。

2. 研究の目的

著者は情報セキュリティ人材の不足という課題の解決を目的とし、情報セキュリティ分野の中でも特に Web アプリケーション脆弱性診断（Web アプリケーション（以下、Web アプリ）に潜む脆弱性を検出し、ユーザに対策を提案する業務）をエンジニアに代わり遂行する脆弱性スキャナ「SAIVS」の研究開発を進めている。

著者は約 7 年間にわたり脆弱性診断に従事しているが、本業界においても十分なスキルを有したエンジニアは大幅に不足している現状がある[6]。なお、脆弱性診断では脆弱性を機械的に検出するツール（以下、脆弱性スキャナ）が使用されることが多く、診断作業の効率化に大きく貢献している[7]。しかし、脆弱性スキャナはエンジニアによる事前の設定や検出した脆弱性の精査等が必要であり、全ての診断作業を自動化することはできない。また、検出できる脆弱性はシグネチャ（脆弱性の検査パターンファイル）で定義したものに限られるため、Web アプリに存在し得る全ての脆弱性を検出することは困難である。

一方、著者が開発を進めている SAIVS は、複数の機械学習アルゴリズムを使用することで、人間による作業を一切必要とせず、Web アプリに存在し得る全ての脆弱性を自動的に検出することを最終目標としている。これが完成した場合、脆弱性診断業界における人材不足の解決に大きく寄与することが期待される。

3. SAIVS の概要

著者は 2016 年に SAIVS のベータ版を開発し[8]、現在までに様々な機能を追加してきた。現在の SAIVS は大きく以下二つの能力を有している。

- Web アプリのクローリング
- 脆弱性の検出

脆弱性診断では、対象の Web アプリに潜む脆弱性を網羅的に検出するために、Web ページを網羅的にクローリングする必要がある。このためには、静的ページは勿論のこと、ログインやユーザ登録といった動的ページも正確にクローリングする必要がある。また、クローリングしながら様々な検査を行い、脆弱性を誤りなく検出する必要がある。

現在の SAIVS は、シンプルな Web アプリにおいて、ログインやユーザ登録等の動的ページを人間と同じようにクローリングすることが可能である。具体的には「ログインページに遷移したが、アカウントを持っていない。よって、先にユーザ登録ページでアカウント作成する。その後、作成したアカウントを使用してログインし、ログイン後のペ

ージをクローリングする。」といった人間の思考を模したクローリングを行うことが可能である。

また、脆弱性の検出においては、Web アプリの挙動を観察し、熟練したエンジニアのように、少ない試行回数で効率よく反射型のクロスサイトスクリプティング（以下、RXSS）を検出することが可能である。これは、Web アプリに渡されるパラメータ値が（HTTP レスポンス上に）出力される箇所を観察し、HTML 構文及び JavaScript 構文の文脈に合致した HTML タグ及び JavaScript を挿入することで、最少 1 回の試行で RXSS を検出することを実現する。また、Web アプリ側でエスケープ処理等のパラメータ値検証を行っている場合は、この検証機構を回避する検査値を自ら考案し、RXSS を検知することが可能である。このように、熟練したエンジニアが脆弱性診断を行う際の思考パターンを模した検査を行うことが可能である。

本稿では、上述した人間の思考パターンを模した「クローリング」と「脆弱性の検出」を実現するための手法の解説と、検証実験の結果を報告する。

4. SAIVS の実現手法

著者は複数の機械学習アルゴリズムを組み合わせて使用することで、人間のような「クローリング」と「脆弱性の検出」を実現した。以下、各能力の実現手法を解説する。

4.1 クローリング

クローリングとは、Web アプリ上のリンクやフォームを基に次ページへと遷移していく行為である。一見すると簡単そうに思えるが、意外にも複雑な思考が必要であり、機械で実現するためには幾つもの技術的な障壁を乗り越える必要がある。例として、Listing 1 の FORM タグを考える。

Listing 1　FORM タグの例

```
1  <form action="/cyclone/sessions" method="post">
2   <label for="email">Email</label>
3   <input id="email" name="email" type="text" />
4   <label for="pass">Password</label>
5   <input id="pass" name="pass" type="password" />
6   <input name="fin" type="submit" value="Sign in" />
7  </form>
```

この FORM から次のページ（/cyclone/sessions）に正しく遷移するためには、INPUT タグで指定されたフィールドに、Web アプリが期待している値を入力しなければならない（行 3、5）。例えば Email には E メールアドレス、Password にはパスワードを入力する必要がある。

また、「Sign in」という記述から（行 6）、このページはログインページであると推測できる。よって、事前にログインアカウントを用意する必要があるが、ログインアカウントを持っていない場合は、先にユーザ登録機能でログインアカウントを作成する必要がある。

さらに、フィールドに適切な値を入力したつもりでも、何らかの理由によりエラーメッセージが表示される可能性もある。この場合は、画面遷移に失敗したことを認識し、フィールドに別の値を入力し直す必要がある。

以上のことから、SAIVS が正しくクローリングを行うためには、最低限以下の要件を実現する必要がある。

- ページ種別の認識
- （フィールドに対する）最適な入力値の学習
- ページ遷移の成否を認識

本稿では、Table 1 に示す 2 つの機械学習アルゴリズムを使用することで、上記要件の実現を試みた。

Table 1　クローリングに使用した機械学習アルゴリズム

要件	アルゴリズム
ページ種別の認識	ナイーブベイズ[9]
ページ遷移の成否を認識	
最適な入力値の学習	Word2Vec[10]

以下、各要件を実現する手法を解説する。

4.1.1 ページ種別の認識

本稿では、ナイーブベイズを使用して「ページ種別の認識」を実現した。ナイーブベイズはテキスト分類に用いられるアルゴリズムであり、分類対象のテキストを特徴付ける情報を手掛かりに、予め定義しておいた複数のカテゴリにテキストを自動分類することができる。なお、ナイーブベイズは既にセキュリティ分野で利用されており、例えばスパムメールフィルタやウェブ侵入検知[11]等で実用されている。

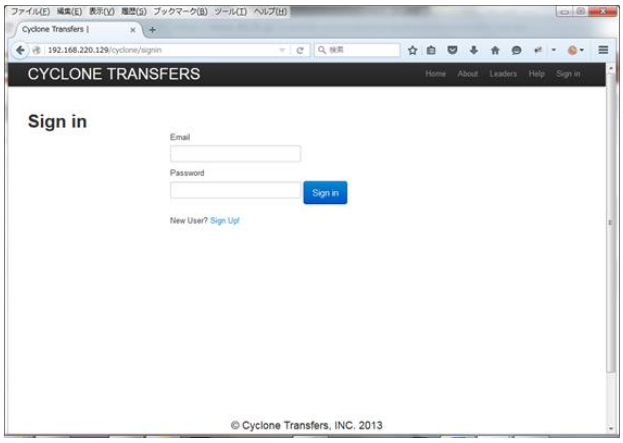

Figure 1　ログインページ

ところで、Figure 1 を見て何を行うためのページだと思うだろうか。大抵の人間は「ログイン」ページであると認識できる。これは過去に見た幾つものログインページの情報を基に、ページ上の「Sign in」「Email」「Password」等のページ種別を特徴付ける情報から「ログイン」と判断しているからである。ナイーブベイズでも同様のロジックでページ種別を認識するが、そのためには人間が無意識に識別しているページ種別を特徴付ける情報を取得する必要がある。すなわち、「Sign in」「Email」「Password」等の文字列を該当ページの HTML からパースして取得する必要がある（Listing 2 の赤字部分）。

Listing 2　ログインページの HTML ソース（抜粋）

```
1  <h1>Sign in</h1>
2  <form action="/cyclone/sessions" method="post">
3    <label for="email">Email</label>
4    <input id="email" name="email" type="text" />
5    <label for="pass">Password</label>
6    <input id="pass" name="pass" type="password" />
7    <input name="fin" type="submit" value="Sign in" />
8  </form>
```

なお、HTML ソースには大量の文字列が含まれているため、より正確にページ種別を認識するためには、ページ種別の特徴付けに寄与しない不要な文字列(ストップワード)を排除する必要がある。本稿では、見出しタグ(<h1></h1>)で囲まれた文字列（行 1）と INPUT タグ付近の文字列（行 3 と 5）のみをパースし、それ以外の文字列は排除した。見出しタグで囲まれた文字列はページの中でも強調されて表示されるため、ページ種別を最もよく表す文字列であると考えられる。また、INPUT タグ付近の文字列はフィールドのラベルに使われるため、ページの目的(ログインなのか、ユーザ登録なのか等）をよく表す文字列であると考えられる。

次に、ナイーブベイズを学習させるために Table 2 に示すカテゴリテーブルを作成した。カテゴリテーブルとは、分類対象のカテゴリ（本項ではページ種別）毎に、各カテゴリを特徴付ける単語を事前に定義したものである。このテーブルを使用してカテゴリ毎に単語の出現確率を学習しておくことで、遷移したページからパースした特徴単語がどのカテゴリに多く含まれているのか、その確率を求めることが可能となる。

Table 2　ページ種別用のカテゴリテーブル（例）

カテゴリ	カテゴリを特徴付ける単語
ログイン	Email, User ID, Password, Sign in, …
ユーザ登録	Email, Password, Confirm, Sign up, …
検索	Word, Text, String, Search, …

なお、本稿で使用したカテゴリテーブルは、著者が無作為に選択した約 50 の Web アプリを訪れ、各ページの HTML ソースからページの特徴付けに寄与すると思われる単語を収集して作成した。

次に、Listing 2 の HTML ソースからパースした単語「Sign in」「Email」「Password」を基に、ページ種別を認識する過程を説明する。カテゴリ分類したいページに対し、各カテゴリにおける各特徴単語の出現確率を使用して、ページの出現確率を計算する。そして、最も出現確率が高いカテゴリを分類結果として返すようにする。なお、本稿ではページの出現確率を「ページ中の各特徴単語の出現確率の積」で求める事にした。Table 3 は Listing 2 の HTML ソースからパースした各特徴単語の出現確率を示している。例えば、カテゴリ「ログイン」において、単語「Sign in」の出現確率は「6%」、「Email」は「9%」、「Password」は「6%」とする。これと同様に、カテゴリ「ユーザ登録」と「検索」における各特徴単語の出現確率を計算する。

Table 3　各特徴単語の出現確率

カテゴリ（ページ種別）	特徴単語の出現確率		
	Sign in	Email	Password
ログイン	6%	9%	6%
ユーザ登録	3%	9%	6%
検索	3%	3%	3%

そして、カテゴリ毎に特徴単語の出現確率の積を取ることで、ページの出現確率を計算する。

ログイン　　: 0.06 * 0.09 * 0.06 * 100 = **0.0324%**
ユーザ登録　: 0.03 * 0.09 * 0.06 * 100 = **0.0162%**
検索　　: 0.03 * 0.03 * 0.03 * 100 = **0.0027%**

カテゴリ毎の出現確率「0.0324」「0.0162」「0.0027」を比較すると、カテゴリ「ログイン」の出現確率が最も大きいことが分かる。この計算結果より、単語「Sign in」「Email」「Password」を含んだ Listing 2 のページは「ログイン」であると分類することができる。

このようにナイーブベイズを使用することで、ページ上の情報を手掛かりに、人間と同じようにページ種別を認識することが可能となる。

4.1.2 ページ遷移成否の認識

「ページ遷移成否の認識」も「4.1.1 ページ種別の認識」と同じ手法で実現することができる。Figure 2 はログイン失敗時に表示されたページを示している。

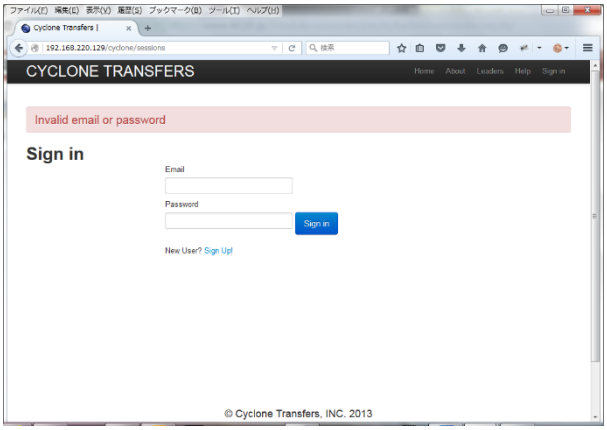

Figure 2　ログインに失敗した様子

大抵の人間は遷移失敗を特徴付ける文字列「Invalid email or password」から、ログインに失敗したことを認識できる。なお、ログイン試行前後の HTML ソース上では、Listing 3 の文字列が差分として現れる。

Listing 3　ログイン失敗時の HTML ソース

```
<div>Invalid email or password </div>
```

よって、Listing 3 の赤字部分のような遷移成否を特徴付ける単語を基にナイーブベイズでカテゴリ分類（ここでは「成功」と「失敗」に分類）することで、遷移に失敗した

か否かを認識することができる。

このように、ナイーブベイズを使用することで、遷移試行後にページ上に現れた情報を手掛かりに、ページ遷移の成否を認識することが可能となる。

4.1.3 最適な入力値の学習

本稿では、自然言語処理で使用される Word2Vec と上述したナイーブベイズを使用することで、最適な入力値の学習を実現した。

通常、フィールドには英字や数字から構成される文字列（名前や住所等）や、英字・数字・記号から構成される文字列（パスワード、E メールアドレス等）が入力される。人間が各フィールドに文字列を入力する際、フィールド傍に記載されているフィールドの意味を表すラベル（ID、Password、Name 等）を参考にして入力文字列を判断しているが、機械はラベルの意味を解釈することは困難である。当然ながら、事前に各ラベルに対応する入力文字列を定義しておくことで、機械的に入力文字列を決定することも可能であるが、ラベルは Web アプリ毎に異なることが多いため、あらゆるラベルに対応する入力文字列を事前に定義しておくことは現実的ではない。

そこで本稿では、各フィールドに総当たりで事前に定義した入力値を与えて遷移を試行し、遷移に成功した場合の入力値とラベルのセットを保存することにした。そして次回以降、新たなフィールドが現れた場合、フィールドのラベルに対応する保存された入力値を使用するようにした。なお、遷移可否の判定には、「4.1.2 ページ遷移成否の認識」のナイーブベイズを使用した。フィールドに入力する入力値の一例を Table 4 に示す。

Table 4　入力値例

入力タイプ	入力値
英字	abc, abcdef, aBc, aBcdEf, …
数字	123, 12345, 123456, 123456789, …
英数字	abc123, 123abc, aBc123, 1a2b3c, …
英数記号	abc!, abc123!, abc#123!abc, a!1b!, …
E メールアドレス	abc1@mbsd.jp, abc2@mbsd.jp, …

ここで、E メールアドレスはログイン ID として使用されることが多く、Web アプリ側で重複チェックが行われることが多い。よって、E メールアドレスのローカル部（@より前の部分）は、使用する度にランダムに変化するようにした。また、本処理を効率よく行うため、Web アプリ側で予め値が入力されているフィールドや、入力値が事前に定義されているセレクトボックスやラジオボタン等は、総当たり入力の対象外とした（hidden 属性が付いたフィールドも同様）。言い換えると、INPUT タグの TYPE 属性が「text」「password」のフィールドのみを総当たり入力の対象とした。

このように、各フィールドに対して総当たりで様々な値を入力し、遷移成功時のラベルと入力値を保存していくことで、様々なフィールドに対する最適な入力値を取得することが可能となる。

SAIVS の訓練

上述の処理は多数の遷移試行を必要とするため、脆弱性診断本番時に新たなフィールドが現れる度に同処理を行うと、クローリング効率が著しく悪化する。そこで本稿では、本番前に SAIVS の学習を行い、あらゆるフィールドに対する最適な入力値を事前に学習しておくことにした。

ここで、事前に学習を行うためには様々なフィールドを含んだ Web アプリが必要になるが、本稿では OWASP Broken Web Apps[12]の「BodgeIt」「Peruggia」「WackoPicko」「OrangeHRM」を使用して訓練を行った。なお、Web アプリが異なると各フィールドに対する入力値も異なる可能性があるが、ユーザ登録や検索といった一般的なフィールドの入力値は Web アプリ間で共通していることが多いため、一つの Web アプリで行った学習結果は、他の Web アプリでも再利用できると考えた。

Table 5 は SAIVS が訓練により学習した各フィールドに対する入力値の一例である。例えば、フィールドのラベルが「ID」の場合は、「abc」や「abcdef」等の文字列を入力することで、訓練時に正しく遷移できたことを示している。この訓練により、ラベル「ID」の場合は「abc」や「abcdef」、ラベル「Email address」の場合は「abc@mbsd.jp」等の入力値を選択できるようになった。

Table 5　各フィールドのラベルに対する入力値（例）

ラベル	遷移成功時の入力値
ID	abc, abcdef, aBc, aBcdEf, …
Password	abc123!, 123abc!, !#$%&a1, aaa, …
First Name	abc, abcdef, aBc, aBcdEf, …
Email address	abc@mbsd.jp, abc123@mbsd.jp, …
Username	abc, abcdef, aBc, aBcdEf, …

ラベルの不一致問題

上述したように、SAIVS は訓練で学習したフィールドのラベルに対する入力値を用いて本番時に遷移を試行する。このため、訓練時に存在しなかったラベルが本番時に現れた場合、訓練で学習した入力値を利用できなくなる問題が生じる。

本稿ではこの問題を解決するために、本番時と訓練時のラベルが一致しない場合は、各ラベルの意味の類似度を計算し、類似度が高い場合は同じラベルと見なすことにした。本稿では類似度の計算に、Word2Vec を使用した。Word2Vec は単語間の意味の類似度を計算し、類似度が 1 に近い単語ほど類似していると見なす。例えば、「E-mail」と「email」の類似度が「0.95」だった場合、この二つの単語は同じものと見なす。なお、Word2Vec は事前に学習させなければ類似度を適切に計算することはできない。そこで本稿では、様々な話題（Computer, Science, Windows 等）に関するニュースグループ記事を纏めたデータセット「The 20 Newsgroups[13]」を用いて Word2Vec の学習を行った。このデータセットには、Web アプリのフィールドのラベルとして頻繁に使用される単語（ID, Password, Email 等）を多く含んだ記事が収録されているため、Word2Vec の学習に最適であると考えた。　Table 6 と Table 7 は、学習済みの Word2Vec が計算した、ラベル「E-mail」と「name」に対する類似単語の一例を示している。

Table 6　ラベル「E-mail」の類似単語一覧

類似単語	類似度
email	0.956302
mail	0.927386
E-mail	0.900249
address	0.893337
reply	0.865438

Table 7　ラベル「name」の類似単語一覧

類似単語	類似度
names	0.962508
username	0.939661
nickname	0.933694
naming	0.898254
surname	0.863966

例えば、本番時に現れたラベル「E-mail」が訓練時に存在しなかった場合、Word2Vec で「E-mail」の類似単語一覧を出力する（Table 6）。そして、一覧内に訓練時に学習したラベルが存在した場合（ここでは「email」とする）、保存しておいた「email」に対応する入力値（abc@mbsd.jp 等）をフィールドに入力する。ここで、類似単語一覧の下位ほど類似度は低くなり、単語間の意味も異なってくる。そこで本稿では、類似単語一覧の内、類似度が 0.7 以上の単語のみを使用することにした。

上述のように、ラベル間の類似度を計算することで、本番時に未知のラベルが現れた場合でも、訓練時の学習結果を最大限利用することが可能となる。なお、類似単語一覧に、訓練時に学習したラベルが存在しない場合は、予め定義した適当な値を使用する。なお、ラベルは単語ではなく文字列として表現される可能性もある。この場合、Word2Vec は使用できないため、他の手法（Doc2Vec[14]等）を採用する必要があるが、これは今後の課題とする。

4.1.4 クローリングの検証

ここでは上述した手法を使用し、実際にクローリングした検証結果を示す。クローリングの検証に使用したテスト環境を Table 8 に示す。

Table 8　クローリングのテスト環境

対象サイト	OWASP Broken Web Apps - Cyclone
対象ページ	ログイン　：http://xxx/cyclone/sessions
	ユーザ登録：http://xxx/cyclone/users
	検索　　：http://xxx/cyclone/search

この Web アプリには「ログイン」「ユーザ登録」「検索」等の動的ページが含まれている。ログインはログイン認証を行うためのページ、検索はログイン後にのみアクセス可能なユーザ検索ページである。なお、初期状態ではログインアカウントを持っていないため、ログイン後のページに遷移するためには、予めユーザ登録でアカウントを作成する必要がある。よって、SAIVS は現在遷移しているページ種別を認識し、ログインなのかユーザ登録なのかを判断する必要がある。Figure 3 にユーザ登録ページを示す。

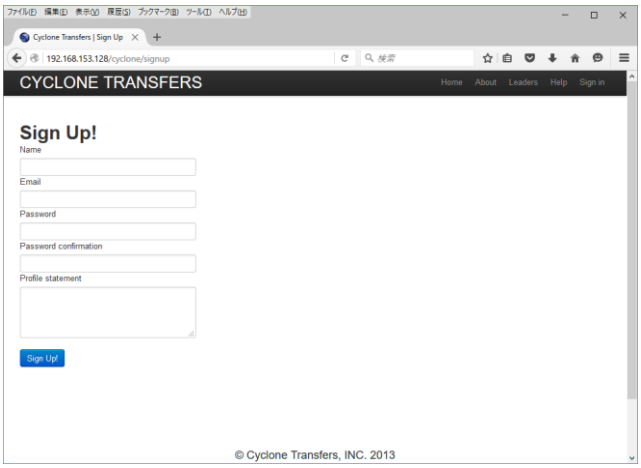

Figure 3　ユーザ登録ページ

Figure 3 から分かるように、ユーザ登録を行うためには、ラベル「Name」や「Email」及び「Password」のフィールドに最適な値を入力する必要がある。よって、SAIVS は各フィールドに対する最適な値を訓練結果に基づいて入力する必要がある。仮に何らかの理由で遷移に失敗した場合は、遷移に失敗したことを認識し、再度別の入力値を用いて遷移を試行する必要がある。

以上から、本テスト環境を用いて SAIVS のクローリング検証を行うことで、クローリングの三要件である以下を満たすことができるのか確認することができる。

- ページ種別の認識
- 最適な入力値の学習
- ページ遷移の成否を認識

検証の様子を Movie 1 に示す。

Youtube：https://www.youtube.com/watch?v=aXw3vgXbl1U

Movie 1　クローリング検証の様子

動画の左上のコンソールは SAIVS を実行しているコンソール、右上のブラウザは SAIVS がクローリングしている様子を分かり易くするために、SAIVS が訪れているページの HTML を Selenium で描画している。また、動画下部の Burp Proxy HTTP History は、SAIVS のアクセスログをリアルタイムに表示している。

Movie 1 から分かるように、初期状態ではアカウントを持っていないため、ログインを後回しにしてユーザ登録（`/cyclone/users`）に遷移している。このことから、SAIVS は正しくページ種別を認識していることが分かる（0:07～0:36）。ここで、コンソール上に表示されている「GREEDY」は、訓練で学習した結果を基に、ユーザ登録を正しく行うための最適な入力値を選択していることを意味している。そして、正しくアカウントを作成した後にログインを行い、ログイン後の検索（`/cyclone/search`）に遷移している（0:37～）。このことから、SAIVS はユーザ登録時の各フィールドに Web アプリが想定している最適な値を入力できていることが分かる。そして最後に、ログイン（`/cyclone/sessions`）の遷移を行い（0:59～1:20）、全

対象ページのクローリングを終了している。
この検証結果から、SAIVS はページ種別を適切に認識し、自ら各フィールドに対する最適な入力値を選択してアカウントを作成し、ログイン後のページに遷移できることが示された。

このように、複数の機械学習アルゴリズムを組み合わせることで、人間のようなクローリングを実現可能であることが示された。

4.2 脆弱性の検出

本稿における「脆弱性の検出」は、下記の実現を目標にしている。

「少ない手数で効率よく脆弱性を検出する。」

従来の脆弱性スキャナのように、多数のシグネチャを使用して脆弱性を検出するのではなく、Web アプリの挙動を観察しながら少ない試行（理想は 1 回）で脆弱性を検出することを目指している。なお、Web アプリの脆弱性は多種多様であるが、本稿では RXSS を対象にする。

4.2.1 RXSS とは

実際に RXSS の検査時のログを用いて説明する。
例えば、「http://xxx/case3/?input=testData」にアクセスすると、Listing 4 のレスポンスが返された場合を考える。

Listing 4　Web アプリのレスポンス

```
<input type="text" value="testData">
```

パラメータ「input」の値「testData」が、INPUT タグの VALUE 属性値に出力されていることが分かる。 今度はパラメータ「input」に以下の値を入力する。

Listing 5　パラメータにスクリプトを入れた様子

```
http://xxx/case3/?input="/><script>alert('XSS')
;</script>
```

すると、Listing 6 のレスポンスが返されたとする。

Listing 6　スクリプトが挿入された様子

```
<input type="text" value=""/><script>alert('XSS'
);</script>">
```

入力した「"/>」によって VALUE 属性値と INPUT タグが閉じられ、後続の SCRIPT タグ「<script>alert('XSS')
;</script>」が挿入されていることが分かる。これにより、ユーザが入力した任意のスクリプト「alert('XSS')」を実行することができる。なお、実際に RXSS を悪用する場合は、正規ユーザを Listing 5 のような細工したリクエストが自動送信される罠ページに誘導し、ユーザ自身で細工リクエストを送信させるようにする。その結果、ユーザのブラウザ上で任意のスクリプトが実行される。
以上のように、RXSS では HTML 構文に合致した HTML タグ及び JavaScript を入力することで、任意のスクリプトを実行させることができる。このことから、RXSS を検出するためには、「HTML 構文の理解」が必須になる。また、入力値は JavaScript 内に出力されることもあるため、「JavaScript 構文の理解」も必須になる。

次に、もう少しセキュアな Web アプリを考える。Listing 5 と同じように、パラメータに以下の値を入力する。

Listing 7　パラメータにスクリプトを入れた様子

```
http://xxx/case4/?input="/><script>alert('XSS')
;</script>
```

すると、Listing 8 のレスポンスが返されたとする。

Listing 8　少しセキュアな Web アプリのレスポンス

```
<input type="text" value=""/> alert('XSS');">
```

入力した SCRIPT タグ「<script></script>」が Web アプリ側で削除されていることが分かる。つまり、スクリプトを実行することはできない。このように、少しセキュアな Web アプリでは、SCRIPT タグを削除するといった入力値の検証機構が存在することが分かる。この検証機構を回避してスクリプトを実行させるためには、SCRIPT タグを使わずにスクリプト実行が可能な検査値を使用する必要がある。そこで、今度は以下の値を入力する。

Listing 9　検証処理を回避するスクリプトを入れた様子

```
http://xxx/case4/?input="onmouseout=alert('XSS'
)"
```

すると、Listing 10 のレスポンスが返されたとする。

Listing 10　スクリプトが挿入された様子

```
<input type="text" value=""onmouseout=alert('XSS
')"">
```

入力した「"」によって VALUE 属性値が閉じられ、後続のイベントハンドラ（onmouseout）が挿入されていることが分かる。これにより、ブラウザに表示された入力フォームからマウスポインタが外れたタイミングで任意のスクリプト「alert('XSS')」が実行される。このように、Web アプリ側に検証機構が存在する場合でも、検証機構を回避する検査値を使用することで、任意のスクリプトを実行させることができる。すなわち、RXSS を検出することができる。
以上の結果から、RXSS を検出するためには、「検証機構の回避」も必須になる。

纏めると、RXSS を検出するためには、最低限以下三つの要件を実現する必要がある。

- HTML 構文の理解
- JavaScript 構文の理解
- 検証機構の回避

以下、各要件を実現する手法を解説する。

4.2.2 RXSS 検出の実現方法

本稿では、Table 9 に示す 3 つの機械学習アルゴリズムを使用することで、上記要件の実現を試みた。

Table 9　RXSS 検出に使用した機械学習アルゴリズム

要件	アルゴリズム
HTML 構文の理解	LSTM[15]
JavaScript 構文の理解	
検証機構の回避	多層パーセプトロン[16] Q 学習[17]

以下、各要件を実現する手法を解説する。

HTML 構文及び JavaScript 構文の理解

LSTM はニューラルネットワークの一種であり、従来のニューラルネットワークでは学習が困難であった時系列データ（動画、音楽、音声等）を学習することが可能である。時系列データを学習するためには長期的・短期的なデータ間の依存関係（例：動画の冒頭と序盤、そして終盤との関係等）を考慮する必要があるが、LSTM はこれを可能にする。この特徴を利用して、音声認識や音楽生成[18]等で実用されている。また、文章も一つひとつの単語を一つの時間と捉えると、時系列データとして考えることができる。この特徴を利用して、Linux のソースコードを LSTM で学習させることで、C 言語風のソースコードを自動生成する検証事例[19]がある。これは、学習に使用した C 言語のソースコード中の各文字の依存関係を LSTM に学習させ、LSTM にソースコード生成の起点となる文字（以下、シード）に続く文字を次々と推測させることで実現している。

本稿では、上述した LSTM の文章生成能力を「HTML 構文の理解」「JavaScript 構文の理解」の実現に利用する。利用例の解説として、Listing 11 のレスポンスを考える。これは入力した値が TEXTAREA タグで囲まれた箇所に出力された様子を示している。TEXTAREA タグで囲まれているため、タグに囲まれた箇所にスクリプトを挿入してもスクリプトを実行することはできない。

Listing 11　TEXTAREA タグ内に出力された様子

```
<textarea name="in" rows="5" cols="60">testData</
/textarea>
```

熟練したエンジニアであれば、Listing 11 のレスポンスを観察することで、「TEXTAREA タグを閉じた後にスクリプトを挿入すれば、スクリプトが実行できる。」と考えることができる。そこで、次に以下の値を入力する。

Listing 12　TEXTAREA タグを閉じる文字列を入れた様子

```
http://xxx/case5/?input=</textarea><script>aler
t(‘XSS’);</script>
```

すると、Listing 13 のレスポンスが返されたとする。

Listing 13　スクリプトが挿入された様子

```
<textarea name="in" rows="5" cols="60"></textare
a><script>alert('XSS');</script></textarea>
```

入力した「</textarea>」によって TEXTAREA タグが閉じられ、「alert('XSS')」を実行することができる。このような検査を行うためには、入力値の出力箇所の前方に TEXTAREA タグの始点（<textarea name=…）があることを観察し、これを HTML 構文に則り TEXTAREA タグの終点（</textarea>）で閉じることに気付く必要がある。

本稿では、「<textarea name=…>の後に</textarea>を入れる。」という、エンジニアが自然に行っている思考を LSTM で実現する。なお、RXSS が発現するパターンはタグ内の属性値やコメント内、FRAMESET タグ内、JavaScript 内等、数多く存在する。このため、LSTM はあらゆるパターンの HTML 構文及び JavaScript 構文を理解する必要がある。そこで本稿では、Listing 14 に示すデータを使用し、LSTM に「HTML のタグや属性等の組み合わせ及び順序関係」を学習させた。なお、このデータは実際にインターネット上で使用されている HTML 構文を無作為に約 2 万ページ分（12,000 パターン）収集したものであり、様々な HTML 構文が含まれている。

Listing 14　HTML 構文の学習データ（抜粋）

```
1  <abbr class="" data-utime="" title=""></abbr>
2  <input name="" type="" value=""/>
3  <input id="" onclick="" src="" type=""/>
4  <video autoplay="" loop="" muted=""></video>
5  <video src='' ></video>
```

なお、LSTM の学習時間はデータ量に比例するため、学習に寄与しない属性値やコンテンツは学習データから排除し、学習効率の向上を図った。また、インターネット上から無作為に収集したデータのため、誤った構文が含まれていることも考えられるが、全体のデータ量と比較すると極めて少ないため、学習結果には影響はないと考えた。

Listing 14 の学習データを用いて LSTM を学習させることで、LSTM にシードを与えると、それに続く最適な HTML 構文を自動生成できるようになる。Table 10 に、学習済みの LSTM に対して様々なシードを与えた結果を示す。

Table 10　LSTM が生成した HTML 構文（一例）

シード	LSTM が生成した構文
<textarea cols="60">	</textarea>xxx
<!-- mbsdtest	-->xxx
<input type="" value="	">xxx

※「xxx」は適当な文字列。

Table 10 から分かるように、シードに対応する後続の HTML を適切に生成できていることが分かる。なお、RXSS を検出する場合は、入力値が出力される個所を起点にし、前方の何文字かの文字列を LSTM にシードとして与える。そして、LSTM が生成した HTML 構文の後に任意のスクリプトを付与したパラメータ値を再度 Web アプリに送信することで、HTML 構文に合致した HTML 及び JavaScript を

挿入することが可能となる。

また、JavaScript 構文も同様に学習させる。Listing 15 は JavaScript 用の学習データを示している。この学習データも HTML と同様にインターネットから収集したものであり、約 1 万ページ分の JavaScript 構文が含まれている。

Listing 15　JavaScript 構文の学習データ（抜粋）

```
1  _satellite.pageBottom();']
2  (function(window) {
3    var _gaq = _gaq || [];
4    var methods = ['log', 'dir', 'trace'];
5    if (typeof window.console === 'undefined') {
6      window.console = {};
7    }
```

Table 11 は、学習済みの LSTM に対して様々なシードを与えた結果を示している。

Table 11　LSTM が生成した JavaScript 構文（一例）

シード	LSTM が生成した構文
var hoge = ['log', '	red’];¥r¥n xxx
/* mbsdtest	*/ xxx
function(){	xxx }¥r¥n

HTML 構文と同様に、シードに対応する後続の JavaScript 構文を適切に生成できていることが分かる。

このように、実際に使用されている様々な HTML 構文及び JavaScript 構文を LSTM の学習データとして利用することで、RXSS を自動検出するための要件「HTML 構文の理解」「JavaScript 構文の理解」を実現することができる。

次節は、三番目の要件である「検証機構の回避」を実現する手法を解説する。

検証機構の回避

本稿では、多層パーセプトロン（以下、MLP）と Q 学習を組み合わせたモデルを構築し、検証機構の回避を実現した。本稿で構築したモデルを Figure 4 に示す。本モデルは、Web アプリへの入力値が「出力される個所（以下、出力箇所）」と「検証機構の内容」を入力に取り、それに対応する一つの検査値を出力する。そして、選択した検査値がどの程度 RXSS の検出に寄与したのかを Q 学習で評価することで、最適な検査値を学習していく。

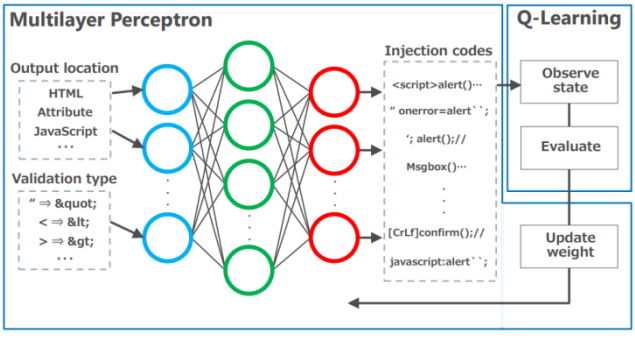

Figure 4　検証機構を回避するモデル

本稿で構築した MLP は三層構造になっており、入力層が 6 ノード（青の丸）、中間層が 100 ノード（緑の丸）であり、出力層（赤の丸）は検査値パターン数により変化する（現在は 20 ノード）。なお、現時点では MLP に入力する出力箇所を以下の 5 種類に限定している。

- 「"」で囲まれた属性値（<～hoge="xxx">）
- 「'」で囲まれた属性値（<～hoge='xxx'>）
- 何にも囲まれていない属性値（<～hoge=xxx>）
- JavaScript 内（<script>xxx</script>）
- HTML タグの外側（<～>xxx</～>）

また、検証機構の内容も以下の 5 種類の組み合わせに限定している。

- 「"」の実体参照への変換、または削除
- 「'」の実体参照への変換、または削除
- 「<」の実体参照への変換、または削除
- 「>」の実体参照への変換、または削除
- 「alert();」の削除

なお、実際に MLP に入力する情報は、各種類に割り当てたユニークなインデックス値としている。

出力層の各ノードは各検査値に紐付いており、MLP に入力された「出力箇所と検証機構の内容」に対してどれか一つの検査値が選択されるようになっている。なお、全く意味のない検査値（JavaScript 内にも関わらず HTML タグを挿入する等）が選択されるのを防ぐため、出力箇所（HTML または JavaScript）に応じて、選択できる検査値を分離している（Table 12）。なお、この検査値は、著者が代表的な検査値を事前に定義したものである。

Table 12　出力箇所毎の検査値（一例）

出力箇所	検査値
「"」で囲まれた属性値 「'」で囲まれた属性値 引用符のない属性値 HTML タグの外側	"event handler "><sCriPt>xxx "><img src=xxx 'event handler '><sCriPt>xxx '><img src=xxx <svg/onload=alert()>

JavaScript 内	";alert();//
	[CR][LF]alert();
	';alert();//
	¥";alert();//

上述した MLP を使用することで、以下に示す人間のような思考を実現することができる。

「Web アプリへの入力値が INPUT タグの VALUE 属性値に出力されているが、「'」「>」「<」はエスケープされる。この場合、「"onmouseout=alert();」が使える可能性が高いので、この検査値を試行する。」

ここで、MLP は教師あり学習モデルであり、学習させなければ最適な検査値を選択することはできない。よって、何らかの方法で MLP を学習させる必要がある。通常、MLP を学習させる場合は、教師データを使用して MLP の重みを最適化していくが、今回のタスクのように無数の「出力箇所と検証機構の内容」に対する検査値を用意することは現実的ではない。

そこで本稿では、MLP に強化学習の一手法である Q 学習を組み合わせ、以下のフローを繰り返し行うことで、MLP の学習を行うことにした。なお、Q 学習は「人間が成功と失敗を繰り返しながら徐々に成功する行動パターンを学習していく」のような学習過程を表現することが可能である。

1. 出力箇所と検証機構の内容を MLP に入力。
2. MLP が何らかの検査値を選択。
3. 2 で選択された検査値を用いて、Web アプリにリクエスト送信。
4. レスポンスを解析し、RXSS の検出有無を確認。
5. 検出有無に応じて Q 値を更新し、更新前後の Q 値の誤差を算出。RXSS 検出時は Q 値を大きく更新、未検出の場合は小さく（または負）更新。
6. 誤差が最小になるように、MLP の重みを更新。

ここで、本稿における Q 値は、検査値の価値を表す。上述の学習においては、RXSS 検出に寄与する検査値ほど Q 値が大きくなるように設計されている。なお、初期状態では Q 値はランダムに初期化されている。

このような学習方法は Deep Reinforcement Learning と呼ばれるが、本稿では Google DeepMind 社の DQN（Deep Q-Network）のアルゴリズム[20]を参考にした。上述の処理を 100 回程度繰り返すことで、徐々に MLP の学習が進んでいく。この学習の仕組みを SAIVS に組み込むことで、学習初期はランダムな検査値を選択していた SAIVS が徐々に賢くなっていき、学習の終盤には Web アプリの検証機構を回避する検査値を優先的に選択できるようになる。

このように、MLP と Q 学習を組み合わせたモデルを使用することで、Web アプリの検証処理を回避可能な検査値を学習することが可能となる。

SAIVS を訓練する

上述の学習は約 100 回の検査試行を伴うため、脆弱性診断の本番時にこれを行うと、診断効率が悪化する。そこで本稿では、事前に SAIVS の訓練を行うことで、本番時に Web アプリの検証機構を回避可能な検査値を即座に選択できるようにした。

ここで、事前に学習を行うためには様々な RXSS のパターンを含んだ Web アプリが必要になるが、本稿では WAVSEP[21]を使用して訓練を行った。WAVSEP には様々な検証機構が実装された RXSS ケースが多く含まれているため、これらのケースに対して SAIVS が繰り返し検査を行うことで、効果的に訓練が行えると考えた。Table 13 に訓練に使用したケースの一例を示す。

Table 13　訓練に使用したケース

WAVSEP - ReflectedXSS GET Input Vector	
Case06	出力箇所：IMG タグの SRC 属性値（"） 検証機構：「<」「>」を実体参照に変換 回避例　："onmouseover=alert(3122);"
Case08	出力箇所：IMG タグの SRC 属性値（'） 検証機構：「<」「>」を実体参照に変換 回避例　：'onmouseover=alert(3122);'
Case10	出力箇所：SCRIPT タグの onClick 値（'） 検証機構：「"」「<」「>」を実体参照に変換 回避例　：';alert(3122);//
Case11	出力箇所：SCRIPT タグの onClick 値（"） 検証機構：「'」「<」「>」を実体参照に変換 回避例　：";alert(3122);//
Case27	出力箇所：JavaScript の単行コメント内 検証機構：コメントアウト 回避例　：[CR][LF]alert(3122);//

各ケースに対して約 100 回の検査を試行し、出力箇所及び検証機構に対する最適な検査値を学習させた。これにより、SAIVS は様々な検証機構を回避する最適な検査値を優先的に選択できるようになった。

ここで、本番時に訓練ケースに含まれていない未知の検証機構に遭遇した場合、SAIVS はこれを回避することはできない。現在の運用では、エンジニアが本パターンの検証を手動で行った上で SAIVS の再訓練を行うことにしている。これにより、次回からは本パターンの検証機構を回避できるようになる。すなわち、検査経験を積み重ねるほど、SAIVS は賢くなっていく。

SAIVS の検査フロー

上述した解説を纏めると、SAIVS は以下のフローで RXSS の検出を試みる。なお、LSTM の学習と検証機構の回避訓練は完了しているものとする。

－ 第 1 次検査 －

1. Web アプリのパラメータに適当な入力値を設定してリクエストを送信。
2. レスポンスを解析し、入力値の出力箇所を特定。
3. 出力箇所を基にシードを抽出。
4. シードに続く HTML 構文及び JavaScript 構文を生成し、末尾に任意のスクリプトを挿入。
5. 4 の文字列を同パラメータに設定して再度リクエストを送信。
6. レスポンスを解析し、RXSS 有無を判定（検出⇒終了、未検出⇒7）。

－ 第 2 次検査 －

7. 出力箇所と検証機構の内容を基に、検証機構を回避できる検査値を選択。
   ※優先順位の高いものから順に選択。
8. 7 の文字列を同パラメータに設定して再度リクエスト送信。
9. レスポンスを解析し、RXSS 有無を判定（検出⇒終了、未検出⇒7）。

上述のとおり、脆弱性の検出は 2 つの工程「第 1 次検査」「第 2 次検査」で行う。第 1 次検査は、入力値が出力される個所を観察した上で、学習済みの LSTM を使用して HTML 構文及び JavaScript 構文の文脈に合致した HTML 及び JavaScript の挿入を試行する。検証機構が存在しない Web アプリの場合は、この 1 回の試行で RXSS を検出する。

第 2 次検査は、検証機構によって RXSS を検出できなかった場合に行われる。出力箇所と検証機構の内容を基に、学習済みの MLP が選択した検証機構を回避する検査値を用いて検査を試行する。訓練で経験済みの検証機構の場合は、この 1 回の試行で RXSS を検出する。1 回目の試行で検出できない場合は、訓練経験を基に優先順位の高い検査値を最大 10 個まで選択して検査を試行する。なお、検査時間短縮の観点から、10 回試行して検出できない場合は、以降の検査試行は行わない。

上述の検査方法により、SAIVS は最少 1 回、最大 11 回の試行で RXSS を検出することが可能である。

4.2.3 脆弱性検出の検証

ここでは「4.2.2RXSS 検出の実現方法」で解説した手法を SAIVS に実装し、実際に Web アプリに対して検査を行った検証結果を示す。

SAIVS vs Webseclab

本稿では、Yahoo 社のエンジニアが開発した脆弱性スキャナ用の検証環境「Webseclab[22]」に対して検査を行った。この検証環境には様々な RXSS のケースが含まれているが、本稿では Table 14 の 4 ケースを対象とした。

Table 14　練習用のケース

Webseclab	
/reflect/full1	出力箇所：BODY タグ内 検証機構：なし
/reflect/textarea1	出力箇所：TEXTAREA タグ内 検証機構：なし
/reflect/onmouseover	出力箇所：INPUT タグの VALUE 属性値 検証機構：閉じタグの削除
/reflect/js4_dq	出力箇所：SCRIPT タグ内 検証処理：なし

なお、Webseclab は画面遷移を必要とせず、対象ケースの URL に直接アクセスすることでテストを行うことができる。以下、ケース毎の検証結果を示す。

Case. [/reflect/full1]

入力値は BODY タグ内に出力され、検証機構は存在しない。正常時のリクエストとレスポンスを Listing 16 に示す。

Listing 16　Case1：正常時のレスポンス

```
[request]
http://xxx/xss/reflect/full1?in=saivs12345
------------------------------------------------
[response]
The value of cgi parameter &quot;in&quot; is:
saivs12345
```

これに対し、RXSS を検出した際のリクエストとレスポンスを Listing 17、Movie 2 に示す。入力値前方の「lasther=''></form>」は、SAIVS が Listing 16 のレスポンスを観察して自動生成した文字列である（Movie 2 の「generating text…」で生成）。

Listing 17　Case1：検査時のレスポンス

```
[request]
http://xxx/xss/reflect/full1?in=lasther=''%3E%3
C/form%3ED0i7Q%22VW53N'nT7t0%3Cscript%3Ealert(3
122);kc5i3%3C/script%3EueFj8
------------------------------------------------
[response]
The value of cgi parameter &quot;in&quot; is:
lasther=''></form>D0i7Q"VW53N'nT7t0<script>aler
t(3122);kc5i3</script>ueFj8
```

Youtube: https://www.youtube.com/watch?v=3RkhSED5DQU

Movie 2　Case1：検査時の様子

本ケースは非常に単純なため、1 回の試行で RXSS を検出できたことが分かる（0:49）。

Case. [/reflect/textarea1]

入力値は TEXTAREA タグ内に出力され、検証機構は存在しない。正常時のリクエストとレスポンスを Listing 18 に示す。

Listing 18　Case2：正常時のレスポンス

```
[request]
http://xxx/xss/reflect/textarea1?in=saivs12345
------------------------------------------------
[response]
<FORM>
<textarea name="in" rows="5" cols="60">saivs1234
5</textarea>
```

これに対して、RXSS を検出した際のリクエストとレスポンスを Listing 19、Movie 3 に示す。入力値前方の「</textarea>」は、SAIVS が Listing 18 のレスポンスを観察して自動生成した文字列である（Movie 3 の「generating text…」で生成）。

Listing 19　Case2：検査時のレスポンス

```
[request]
http://xxx/xss/reflect/textarea1?in=%3C/textare
a%3E7Q7pN%22MBPcc'PA6tz%3Cscript%3Ealert(3122);
WKr8J%3C/script%3EfowCP
------------------------------------------------
[response]
<FORM>
<textarea name="in" rows="5" cols="60">&lt;/textare
a&gt;7Q7pN"MBPcc'PA6tz<script>alert(3122);WKr8J</s
cript>fowCP</textarea>
```


Youtube: https://www.youtube.com/watch?v=6UHbMGdqr_0

Movie 3　Case2：検査時の様子

SAIVS が生成した HTML タグ「</textarea>」によって元から存在する TEXTAREA タグが閉じられ、その後に SCRIPT タグが挿入できたことが分かる。本ケースも 1 回の試行で RXSS を検出できたことが分かる（0:31）。

このように、HTML 構文を理解した上で検査できることが示された。

Case. [/reflect/onmouseover]

入力値は INPUT タグの VALUE 属性値に出力され、また、タグ閉じ（</script>、</option>等）を削除する検証機構が備わっている。正常時のリクエストとレスポンスを Listing 20 に示す。

Listing 20　Case3：正常時のレスポンス

```
[request]
http://xxx/xss/reflect/onmouseover?in=”><script
>alert()</script>
-----------------------------------------------
[response]
Homepage: <input value="><script>alert()" name=
"in" size="40"><BR>
```

SCRIPT タグ閉じ（</script>）が削除されることで、スクリプトが動作しないことが分かる。これに対し、RXSS を検出した際のリクエストとレスポンスを Listing 21、Movie 4 に示す。入力値前方の「"></option><option s」は、SAIVS が Listing 20 のレスポンスを観察して自動生成した文字列である（Movie 4 の「generating text…」で生成）。

Listing 21　Case3：検査時の様子

```
[request]
http://xxx/xss/reflect/onmouseover?in=%22%3E%3C
/option%3E%3Coption%20s%20onmouseover=alert(312
2);//
-----------------------------------------------
[response]
Homepage: <input value="> <option s onmouseover=a
lert(3122);//" name="in" size="40"><BR>
```

Youtube: https://www.youtube.com/watch?v=-r3C1moUVqU

Movie 4　Case3：検査時の様子

Movie 4 を見ると、最初は検査値「"></option><option … <script>alert(3122);…</script>」を用いて検査を行っているが（0:29）、検証機構により RXSS が検出できなかったため、次に検証機構を回避する検査値を次々と試していることが分かる（0:31）。これは過去の訓練を基に優先順位の高い検査値から順に選択している。そして、3 回目の試行において、Listing 21 に示す検査値を用いて検証機構を回避して RXSS を検出している。

このように、Web アプリの検証機構を回避して検査できることが示された。

Case. [/reflect/js4_dq]

入力値は SCRIPT タグ内に出力され、検証機構は存在しない。なお、本ケースは JavaScript 内に入力値が出力されるパターンである。正常時のリクエストとレスポンスを Listing 22 に示す。

Listing 22　Case4：正常時のレスポンス

```
[request]
http://xxx/xss/reflect/js4_dq?in=saivs12345
-----------------------------------------------
[response]
<script language="javascript">
var f = {
        date: "",
        week: "1",
        bad: "saivs12345",
        phase: "2",
```

これに対し、RXSS を検出した際のリクエストとレスポンスを Listing 23、Movie 5 に示す。入力値前方の「6",[CR][LF][SP]」は、SAIVS が Listing 22 のレスポンスを観察して自動生成した文字列である（Movie 5 の「generating text…」で生成）。

※[CR]は復帰、[LF]は改行、[SP]は空白を示す。

Listing 23　Case4：検査時のレスポンス

```
[request]
http://xxx/xss/reflect/js4_dq?in=6%22,%0A%20%20
%20%20%20%20%20%20%20%20%20%20skuI;alert(3122);
//1VU7k
-----------------------------------------------
[response]
<script language="javascript">
var f = {
        date: "",
        week: "1",
        bad: "6",
             skuI;alert(3122);//1VU7k",
        phase: "2",
```

Youtube: https://www.youtube.com/watch?v=Pf2lSB25C3M

Movie 5　Case4：検査時の様子

SAIVS が生成した「",」によって元から存在するオブジェクトリテラルのキー「bad」の値宣言が閉じられ、改行後に「alert(3122);//」が挿入されたことが分かる。そして、本ケースも 1 回の試行で RXSS を検出したことが分かる（0:27）。

このように、JavaScript 構文を理解した上で検査できることが示された。

この検証結果から、SAIVS は HTML 構文及び JavaScript 構文を理解し、検証機構を回避しながら検査を遂行できることが示された。

このように、複数の機械学習アルゴリズムを組み合わせることで、人間のような脆弱性診断を実現可能であることが示された。

4.3 実戦的な検証

最後に実戦的な検証として、Google 社が開発したセキュリティエンジニア向けの練習環境「Google gruyere[23]」に対して検査を行った。なお、本検証において、人間の関与は SAIVS に Google gruyere のトップ URL を与えるのみであり、その他クローリングや検査は全自動で行った。

なお、Google gruyere には様々な機能が備わっているが、本稿では Table 15 に示す 2 つの機能を検証対象とした。

Table 15　検証対象の機能

Google gruyere	
New Snippet	・データ（snippet）登録機能 ・要ログイン ・入力値が BODY タグ内に出力 ・<script>及び</script>を削除
Profile	・プロフィール編集機能 ・要ログイン ・要プロフィール登録 ・入力値が A タグ内の HREF 属性値に出力 ・< 及び >を実体参照文字に変換

Table 15 の通り、Google gruyere は検証対象の機能に遷移するために「ログイン画面⇒ログイン処理⇒ログイン後トップページ…」のように複数の画面をクローリングする必要がある。また、クローリングは単純にリンクを踏むだけではなく、入力フォームに適切な値を入力しなければ次ページに遷移できないケースも存在する（ログインやプロフィール登録等）。更に、ログインするためには事前にユーザ登録機能でアカウントを作成する必要がある。本検証においては、「4.1 クローリング」で解説した手法を用いて複雑なクローリングが行えるのか確認した。

また、Web アプリへの入力値は BODY タグや A タグといった様々な箇所に出力され、特定のスクリプト文字列や記号は削除及び実体参照文字に変換される（検証機構）。本検証では、「4.2 脆弱性の検出」で解説した手法を用いて、HTML 構文及び JavaScript 構文の文脈に合致した HTML 及び JavaScript の生成や、検証機構を回避する検査値の選択が適切に行えるのか確認した。

以上から、Google gruyere を用いて SAIVS の実戦的な検証を行うことで、本稿の目的である以下の機能が実現できるのか確認することができる。

- Web アプリのクローリング
- 脆弱性の検出

Movie 6 に検証時の様子を示す。

Youtube: https://www.youtube.com/watch?v=N5d9oM0NcM0

Movie 6　検査時の様子

動画の左上のコンソールは SAIVS を実行しているコンソール、右上のブラウザは SAIVS がクローリングしながら検査を行っている様子を分かり易くするために、SAIVS が訪れているページの HTML を Selenium で描画している。また、動画下部の Burp Proxy HTTP History は、SAIVS のアクセスログをリアルタイムに表示している。

Movie 6 から分かるように、初期状態ではアカウントを持っていないため、ログインを後回しにしてユーザ登録（Sign up）を行い（0:20～0:21）、アカウント作成後にログインを行っていることが分かる（0:21～0:22）。そして、ログイン後のページである New Snippet 機能に遷移しながら検査を行い（0:23～0:34）、最後に同じくログイン後の機能である Profile 機能の検査を行っていることが分かる（0:39～2:37）。なお、この一連のクローリングと検査に要した時間は約 145 秒であった。また、検証対象の 2 つの機能で RXSS を検出することも確認された。

以下、各機能に対する検証結果の詳細を解説する。

Case. [New Snippet]

本機能では、入力値は BODY タグ内に出力され、入力値に含まれる「<script></script>」が検証機構により削除される。正常時のリクエストとレスポンスを Listing 24 に示す。

Listing 24　正常時のレスポンス

```
[request]
http://xxx/1142014131/newsnippet2?snippet=saivs
12345
------------------------------------------------
[response]
<td valign='top'>
  <div id='20'>
    saivs12345
```

次に、Listing 25 に RXSS を検出した際のリクエストとレスポンスを示す。

Listing 25　検査時のレスポンス

```
[request]
http://xxx/1142014131/newsnippet2?snippet=widtt
='';%3E%3C/option%3E%3Cs%20onmouseover=alert(312
2)%3E
------------------------------------------------
[response]
<td valign='top'>
 <div id='0'>
   widtt=''></option><s onmouseover=alert(3122)>
```

SAIVS は本ケースの検証機構（<script></script>の削除）を回避する検査値を用いて RXSS を検出できたことが分かる。

Case. [Profile]

本機能では、入力値は A タグ内の HREF 属性値に出力され、検証機構により「<」「>」が実体参照文字に変換される。正常時のリクエストとレスポンスを Listing 26 に示す。

次に、Listing 27 に RXSS を検出した際のリクエストとレスポンスを示す。

Listing 26　正常時のレスポンス

```
[request]
http://xxx/1142014131/saveprofile?action=update
&name=test&oldpw=&pw=&icon=hoge2&color=hoge4&pr
ivate_snippet=hoge5&web_site=saivs12345
------------------------------------------------
[response]
<a href='saivs12345'>Homepage</a>
```

Listing 27　検査時のレスポンス

```
[request]
http://xxx/1142014131/saveprofile?action=update
&name=test&oldpw=&pw=&icon=hoge2&color=hoge4&pr
ivate_snippet=hoge5&web_site='%20onloc=''/%3E%3
Chr%20alt=%20onmouseover=alert(3122)
------------------------------------------------
[response]
<a href='' onloc=''/&gt;&lt;hr alt= onmouseover=a
lert(3122)'>Homepage</a>
```

SAIVS は本ケースの検証機構（< 及び >の実体参照文字への変換）を回避する検査値を用いて RXSS を検出できたことが分かる。

以上の検証結果から、複数の機械学習アルゴリズムを使用することで、人間と同じようにクローリングし、Web アプリの挙動を見ながら検査を行い、少ない手数で RXSS を検出できることが示された。

5. 結論

本稿では、「クローリング」と「脆弱性の検出」を機械学習アルゴリズムによって実現する手法と検証結果を示した。人間の思考パターンを模した機械学習アルゴリズムを複数組み合わせることで、人間と同じような Web アプリのクローリングや RXSS の検出が実現できることを確認した。

今後の課題

完全に人間と同等の能力を持つためには、解決すべき幾つかの課題も存在する。

クローリングについては、業務アプリや EC サイトといった、入力値や遷移の順序が複雑なアプリへの対応や、Ajax/JavaScript 等を使用した複雑なページ遷移への対応及びクローラー防止に使用される CAPTCHA への対応等が必要になる。

脆弱性の検出については、ブラウザ依存や文字コードに起因する RXSS への対応、二重デコードや Unicode エスケープシーケンス変換等の特殊な検証機構への対応等が必要である。

また、本稿では DQN アルゴリズムを使用することで、検証機構を回避する検査値を SAIVS 自ら学習させる手法を採用したが、過去の熟練エンジニアによる脆弱性診断結果を学習データとし、SAIVS に教師あり学習をさせる手法も有効であると考えられる[24]。

診断対象の挙動に応じて、過去の脆弱性診断結果を利用できる場合は教師あり学習モデルを使用し、過去に経験のない未知の脆弱性パターンの場合には、本稿で提案したモデルを使用することで、診断の柔軟性が増すと考えられる。これは今後の課題とする。

現在の SAIVS はベータ版である。今後、より多くの Web アプリを診断できるよう、機械学習を始めとした様々な技術を駆使し、上述の課題を解決していきたいと考えている。そして、SAIVS が人間と同等、もしくは人間をサポートする程の能力を獲得した際には、脆弱性診断業務に投入していきたいと考えている。

参考文献

[1] 警視庁. 2017. 平成 28 年におけるサイバー空間をめぐる脅威の情勢について. https://www.npa.go.jp/publications/statistics/cybersecurity/data/H28cyber_jousei.pdf

[2] 経済産業省. 2015. 情報セキュリティ分野の人材ニーズについて. http://www.meti.go.jp/committee/sankoushin/shojo/johokeizai/it_jinzai_wg/pdf/002_03_00.pdf

[3] 労働政策研究・研修機構. 2016. 人材（人手）不足の現状等に関する調査 http://www.jil.go.jp/press/documents/20160615.pdf

[4] DARPA. 2016. THE WORLD'S FIRST ALL-MACHINE HACKING TOURNAMENT. https://www.cybergrandchallenge.com/

[5] Leyla Bilge, Tudor Dumitras, Symantec Research Labs, 2012. Before We Knew It. An Empirical Study of Zero-Day Attacks In The Real World

[6] 経済産業省. 2015. 脆弱性診断人員に対するスキル評価に係る検討. http://www.meti.go.jp/meti_lib/report/2015fy/000581.pdf

[7] SECTOOL Market. Price and Feature Comparison of Web Application Scanners. http://sectoolmarket.com/price-and-feature-comparison-of-web-application-scanners-unified-list.html

[8] Isao Takaesu, Mitsui Bussan Secure Directions. 2016. Overview of the “Vulnerability Scanning AI” using the machine learning. http://www.mbsd.jp/blog/20160113_2.html

[9] Willi Richert, Luis Pedro Coelho, O’Reilly. 2014. 実践 機械学習システム

[10] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, Google. 2013. Efficient Estimation of Word Representations in Vector Space

[11] 金床，ビットフォレスト. 2014. ベイジアンネットワークを使ったウェブ侵入検知.
https://www.scutum.jp/information/waf_tech_blog/2014/02/waf-blog-034.html
[12] OWASP. OWASP Broken Web Applications Project.
https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project
[13] The 20 Newsgroups data set.
http://qwone.com/~jason/20Newsgroups/
[14] Quoc Le, Tomas Mikolov, Google. 2014. Distributed Representations of Sentences and Documents
[15] The MIT Press. Long Short-Term Memory. 2006.
http://www.mitpressjournals.org/doi/abs/10.1162/neco.1997.9.8.1735
[16] 岡谷貴之，講談社. 2015. 深層学習
[17] 三上貞芳，皆川雅章，森北出版. 2008. 強化学習
[18] Richard Adhikari, Tech News World. 2016. Google's Magenta AI Tickles the Ivories.
http://www.technewsworld.com/story/83567.html
[19] Andrej Karpathy. 2015. The Unreasonable Effectiveness of Recurrent Neural Networks.
http://karpathy.github.io/2015/05/21/rnn-effectiveness/
[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, DeepMind. 2013. Playing Atari with Deep Reinforcement Learning
[21] sectooladdict. WAVSEP.
https://github.com/sectooladdict/wavsep
[22] Yahoo Inc. Webseclab
https://github.com/yahoo/webseclab
[23] Google. Web Application Exploits and Defenses.
https://google-gruyere.appspot.com/
[24] Isao Takaesu, Mitsui Bussan Secure Directions. 2017. Recommender system for Web security engineers - basic level -
http://www.mbsd.jp/blog/20170707.html

About us

三井物産セキュアディレクション株式会社（Mitsui Bussan Secure Directions, Inc.）は、セキュリティ対策を支援する専門企業です。情報漏えい調査、セキュリティ診断、セキュリティ監視、セキュリティコンサルティング、内部統制支援等を提供しています。当社のサービスはアプリケーションセキュリティ、セキュリティ診断、侵入テスト、セキュアプログラミング及び攻撃者の方法論等に関する深い知識と経験を持つ専門家によって提供されています。
URL：http://www.mbsd.jp/

About the author

高江洲 勲（Isao Takaesu）, CISSP
プロフェッショナルサービス事業部
先端技術セキュリティセンター

