Computer Security Symposium 2020
26 - 29 October 2020

Metasploit 攻撃コードに対する網羅的な攻撃パケット生成による侵入検知システムのシグネチャ自動生成

小林 雅季[1,a)]　鐘本 楊[2]　小谷 大祐[1,b)]　岡部 寿男[1]

概要：一般に，脆弱性情報の公表と前後して，PoC コードと呼ばれる，その脆弱性の実現可能性を示すコードが公開される．ここで，PoC コードの開発に多用されるツールとして Metasploit が挙げられる．Metasploit のモジュールとして開発されたコードは容易に悪用できるため，公開後の迅速な対策が必要となる．本研究では，その対策として侵入検知システムのシグネチャを生成し，システムに設定することを考える．そうした対策では，手動でのコードの解析によるシグネチャ作成が行われることが多いが，時間がかかる点が問題となる．そこで，本研究では Metasploit の攻撃コードのうち，HTTP によるものについて、そのコードからシグネチャを自動生成する手法を提案する．提案手法では，まず攻撃コードの抽象構文木を用いた制御パスの全列挙と、それぞれの制御パスに従った攻撃コードの実行によって，網羅的に攻撃パケットを生成する．その後，生成された攻撃パケットから n-gram をベースとしたアルゴリズムによって特徴的な文字列を抽出し，シグネチャを生成する．また，代表的な攻撃コードについて生成されたシグネチャ例を示す．

キーワード：ネットワークセキュリティ，Metasploit，侵入検知システム

Automatic Signature Generation for Intrusion Detection Systems with Exhaustively Generated Packets from Metasploit Attack Script

Masaki Kobayashi[1,a)]　Yo Kanemoto[2]　Daisuke Kotani[1,b)]　Yasuo Okabe[1]

Abstract: In general, before and after the disclosure of a vulnerability, the PoC(Proof of Concept) code is released. Metasploit is a framework often used to develop such code. The PoC code developed as a Metasploit module can be easily abused, so we need prompt countermeasures after released. As a basic countermeasure, we need to investigate such code and define signatures for IDS(Intrusion Detection Systems), but it is time-consuming and problematic. In this paper, we propose a method to automatically generate signatures from Metasploit attack script related to HTTP. In this method, we first enumerate all the control paths using an abstract syntax tree of the script and execute the script corresponding to each path to generate attack packets exhaustively. Then, we use an n-gram-based algorithm to extract the characteristics of the attack conducted by the script and generate signatures. We also show examples of signatures generated for some scripts.

Keywords: Network Security, Metasploit, Intrusion Detection System

1. はじめに

脆弱性情報の公開と前後して，PoC(Proof of Concept) コードと呼ばれる，その脆弱性への攻撃が実現可能なことを示すコードが公開されることが多い．本来は脆弱性の存

[1] 京都大学
Kyoto University
[2] NTT セキュアプラットフォーム研究所
NTT Secure Platform Laboratories
[a)] kobayashi@net.ist.i.kyoto-u.ac.jp
[b)] kotani@media.kyoto-u.ac.jp

在を実証するコードを提示することによってその脆弱性への対策を促すという正当な目的のために公表されるが，その性質上悪用することも容易であるため，何らかの対策が必要である．

そのような PoC コードの開発に関連したツールとして Metasploit[*1]が挙げられる．Metasploit とは，既存の脆弱性を利用した攻撃を実行することによってシステムに存在する脆弱性を検査するオープンソースの Ruby によるツールである．それらの脆弱性への攻撃コードはモジュール化され，簡単に利用することができるようになっており，また，新たな攻撃コード開発のためのライブラリも豊富に用意されている．そのため，公開されている PoC コードが Metasploit のモジュールとして開発されている事も多い．実際，脆弱性に対する PoC コードをまとめたサイト[*2]では，Metasploit のモジュールとして実装された多くの PoC コードが確認できる．また，4 年間のハニーポットによる観測を元にした Metasploit による攻撃動向の調査 [7] では，Metasploit による攻撃コード公開の数日間に渡って，そのコードを利用した攻撃が観測された．ゆえに，Metasploit の攻撃コードによる攻撃に対処する事で，公開された PoC コードによる攻撃への対策の 1 つとなることが期待できる．

本稿では，そのような Metasploit を用いた PoC コードによる攻撃を迅速に検知できるようにするため，攻撃コードから侵入検知システム (IDS) の 1 つである Snort[*3]のルールを自動生成する手法を提案する．本来は様々なプロトコルに対応するべきであるが，今回は Web アプリケーションなどで広く利用されている HTTP による通信を対象にした．先行研究では，Metasploit の攻撃コードによる攻撃への対策として，攻撃コードのシンボリック実行によって Snort のシグネチャを自動生成する手法 [9] が提案されている．しかし，このようなシンボリック実行を実装するためには，対象のツールや言語の関連する関数を拡張する必要があり，その実装・メンテナンスにかかるコストが課題となる．

そこで，本研究では，攻撃コードの抽象構文木を利用して制御パスの全列挙とそれぞれのパスに従った攻撃コードを実行することで．シンボリック実行を模倣し，Snort のルール生成を行う．本手法は 3 つの段階から成る．1 段階目では，Ruby で記述された Metasploit の攻撃コードの抽象構文木を利用することによってその制御パスを全列挙し，それぞれのパスに従った攻撃コードにより集合を構成する．2 段階目では，生成した攻撃コード集合に含まれるコードを Metasploit で実際に実行することによって，実行時に攻撃対象のサーバに送出される HTTP リクエスト列をキャプチャする．3 段階目では，キャプチャした HTTP リクエスト列について，n-gram をベースにした手法によって対象の攻撃の特徴となる部分を抽出し，統合することによって Snort のルールを生成する．また，いくつかのモジュールについて本手法によって生成された Snort のルール例を示し，その特徴と本手法が適用できる攻撃コードの例について考察を行う．

本研究の貢献は次の通りである．

- Metasploit の攻撃コードについて，抽象構文木を用いて全ての制御パスを列挙し，それぞれのパスに従う攻撃コード集合を構成する手法の提案
- 攻撃コード集合に含まれる攻撃コードの実行により生成された HTTP リクエスト列の集合から，n-gram をベースとした手法によって Snort のルールを生成する手法の提案

2. 先行研究

本章では，先行研究の中から本研究で着目した事柄と課題についてまとめる．

2.1 ネットワークフローの分析によるシグネチャ生成

2000 年代初頭に，Code Red や SQL Slammer などのワームの爆発的な流行によって，ネットワークのフローから攻撃の可能性のあるフローを抽出し，それらのフローに共通するペイロードを抽出することで，未知の攻撃に対するシグネチャの自動生成を行おうとする研究が盛んになった．そのような研究として，EarlyBird[8] や，Honeycomb[4]，Autograph[2] などを挙げる事ができる．

それに対し，Polygraph[6] や Hamsa[11] では，従来の手法では難読化されたワームの検知が難しいとしている．そして，そのようなワームを検知するために，難読化が難しい，ソフトウェア固有の脆弱性を攻撃している部分に着目する必要性が議論されている．また，そのようなワームを検知するシグネチャとして，複数の文字列の集合からなるシグネチャが効果的であるとした．

しかし，Wang ら [9] が指摘している通り，ネットワークフローの分析によるシグネチャ生成を行うためには，事前に十分な数のフローを収集しておく必要があり，時間がかかってしまう点が難点である．そのため，マルウェアの実行コードやソースコードが入手できた場合については，その利用によるより効率的なシグネチャ生成法を検討する余地がある．

2.2 マルウェアの解析手法

プログラムの実行コードやそのソースコードの解析に関する研究について，その解析手法によって大別して述べる．特にマルウェアにおいては，特定の制御パスにおいてしか発現しない攻撃などが想定されるため，制御パスをどの程度網羅する事ができるかが比較の際に重要となる．

[*1] https://www.metasploit.com
[*2] https://www.exploit-db.com
[*3] https://www.snort.org

2.2.1 静的解析

静的解析とは，プログラム中の定数やプログラム自体の構造に着目することによって，実際に実行することなく，プログラムを解析する手法の総称である．静的解析の最大の長所は，プログラムを形式的に解析しているため，その全ての制御パスを網羅することができる点である．短所としては，関数の呼び出し結果など動的に決定する値が登場すると解析ができなくなってしまう点が挙げられ，この欠点を利用して静的解析を妨げるような手法が存在する [10]．

2.2.2 動的解析

動的解析とは，プログラムを実際に実行することで解析を行う手法の総称である．静的解析と対をなす概念であり，長所として，実行中に決定する値が存在しても問題がない，メモリや外部との通信の様子など実システムの様子も解析できる．などがある．短所としては，単純な解析だけではそのプログラムの制御パスの解析漏れが存在する可能性がある．このような問題に対しては，QEMU[5] を用いた複数パス探索が提案されている．

2.2.3 シンボリック実行

シンボリック実行 [3] では，プログラムに対する入力を具体的な値ではなく，変数として実行する．変数として実行することによって，静的解析における静的に値が決定していない時に解析ができない問題，動的解析における制御パスの網羅性が保証されない問題を解決している．

Wang らは，シンボリック実行を用いて Metasploit の攻撃コードから Snort のルールを生成する手法として、MetaSymploit[9] を提案した。MetaSymploit では、Metasploit の攻撃コードをシンボリック実行し，外部に送出されるパケットをシンボリックなものとして扱うことで，そのコードによる攻撃を網羅的に検知するルールの生成を可能にしている．その評価実験では，検証環境を用意することのできた 45 の攻撃コードに対して，それを検知する Snort ルールを生成することに成功している．しかしながら、Baah ら [1] が指摘している通り，MetaSymploit のようなシンボリック実行を実装するためには，ツールや言語の関数の中で，シンボリックな値を受け取る可能性のあるものを全て拡張する必要があり，実装やメンテナンスの手間がかかってしまう点が課題である．

2.3 先行研究のまとめと課題

先行研究の内容を踏まえて，本研究ではシグネチャとして攻撃パケットから特定の脆弱性を突いている部分を抽出したものを利用する．それに加え，ペイロードの難読化に対応するために，複数の文字列の集合を検知するようなシグネチャの生成を行う．

また，先行研究の課題点として，ネットワークフローベースのシグネチャ生成では大規模なフローの収集が必要であること，また攻撃コードのシンボリック実行によるシグネチャ生成ではその実装コストが高いことが挙げられる．そこで，本研究では，まず MetaSymploit[9] におけるシンボリック実行による手法と同様に，攻撃コードが入手できている場合について効率的なシグネチャ生成手法を提案する．また，シンボリック実行の全パス網羅性を模倣しつつその実装コストを下げる手法として，攻撃コードの抽象構文木に着目した．

3. 提案手法

3.1 提案手法の概要

提案手法の全体像を図 1 に示す．主に 3 つの段階を経る事によって，Metasploit の攻撃コードから Snort のルール生成を行なっている．1 段階目では，Ruby で記述された攻撃コードに対して，その抽象構文木を利用することによって制御パスを全列挙し，それぞれのパスに従った攻撃コードにより集合を構成する．2 段階目では，生成した攻撃コード集合中のコードを Metasploit で実際に実行することによって，それぞれの実行時に送出されるパケットをキャプチャし，HTTP リクエスト列を取得する．3 段階目では，まず，取得した HTTP リクエスト列について，n-gram をベースにした手法によって，それぞれの HTTP リクエストを構成するメソッド名などの各要素の文字列において，文字の出現確率のモデリングを行う．その後，そのモデルを利用することで，そのリクエストに特徴的な文字列を抽出・統合し，Snort のルール生成を行なった．

3.2 制御パスの全列挙と対応する攻撃コード集合の生成

3.2.1 概要

本手法では，攻撃コードの実行パスを形式的に全列挙するために，対象のコードをパースした抽象構文木を利用す

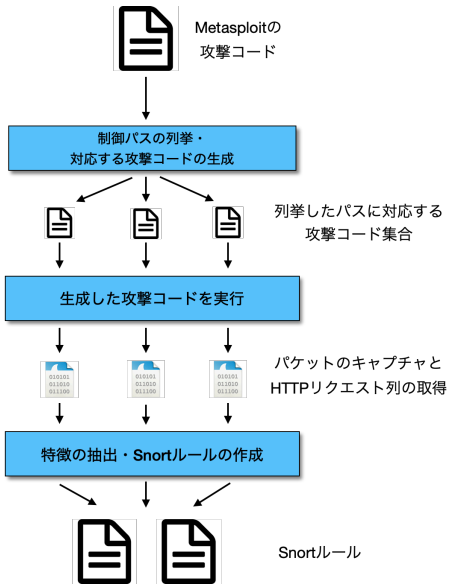

図 1 提案手法の全体像

Fig. 1 The overview of the proposed method

る．if 式などの条件分岐に関するノードに着目する．そのノードが格納している情報を元に，攻撃コードの条件式の部分を書き換えた攻撃コード集合の生成を行う．抽象構文木を利用する事によって，シンボリック実行を用いた Wang らの手法 [9] のような全パス網羅性を模倣しつつ，その実装コストを下げることを目指した．

実装では，Ruby で記述された攻撃コード中の if 式，unless 式，case 式を書き換え，新たな攻撃コード集合の生成を行なった．例として，**図 2** に，if 式を含んだ Ruby コードと，提案手法によってそのコードから生成されるコード集合の一部を示す．ログイン処理など，副作用を持った関数が条件式の部分に現れる場合に備え，条件式を *cond* として $cond \rightarrow (cond; true)$ または $cond \rightarrow (cond; false)$ のように書き換えている．unless 式，case 式についても同様に書き換える．攻撃コードからの抽象構文木の取得に際しては，Ruby の標準ライブラリの機能[*4]を用いた．

3.2.2 アルゴリズム

攻撃コード *code* から生成した抽象構文木について，その根 *root* から深さ優先探索をベースとしたアルゴリズムによって新たなコード集合の生成を行う．抽象構文木の各ノードには，行番号など元のコード上での位置情報と，そのノードが if 式であるなどの，ノードの種類に関する情報が格納されている．アルゴリズムの詳細を，Algorithm1 に示す．まず，攻撃コードをパースすることによって抽象構文木とその根ノード *root* を取得し，また，攻撃コード集合 *codes* を $codes \leftarrow \{code\}$ として初期化する．次に，WALK によって，*root* から抽象構文木中のノード *node* について，再帰的な探索とコードの書き換えを行う．WALK 内では，まず，*node* の種類によって *code* から条件式部分を書き換えたコード集合の生成を行う CONVERT を *codes* に含まれる攻撃コードに対して順次適用し，その結果を *next_codes* に追加する．CONVERT では，*node* に格納されている位置情報を取り出すことで，書き換えが必要な箇所を特定している．その後，*node* から子ノードの集合を取得する CHILDREN によって，再帰的に WALK を適用し，その結果で *next_codes* を更新していく．このような手法によって，*code* 中の制御パスを全列挙し，それぞれのパスに従った攻撃コードにより集合 *codes* を構成した．

```
if login
    print "ok"
end
```

```
if (login; true)
    print "ok"
end
```

図 2 Ruby コード中の if 式の書き換え
Fig. 2 Convertion of if expression in Ruby code

[*4] https://docs.ruby-lang.org/en/2.6.0/RubyVM/AbstractSyntaxTree.html

Algorithm 1 制御パスの全列挙と対応する攻撃コード集合生成のアルゴリズム

Require: *code*: 攻撃コード
Ensure: *codes*: 生成された攻撃コード集合

```
 1: function GENARATE(code)                ▷ 抽象構文木を取得する
 2:     root ← PARSE(code)                 ▷ 攻撃コードを parse
 3:     codes ← {code}
 4:     return WALK(root, codes)
 5: end function
 6:
 7: function WALK(node, codes)             ▷ 再帰的に変換を行う
 8:     if node is Leaf then               ▷ 葉に到達したら終了
 9:         return codes
10:     else
11:         next_codes ← φ
12:         for all code ∈ codes do        ▷ 新たなコードを生成
13:             generated ← CONVERT(node, code)
14:             next_codes ← next_codes ∪ generated
15:         end for
16:
17:         for all child ∈ CHILDREN(node) do ▷ 子ノードを探索
18:             next_codes ← WALK(child, next_codes)
19:         end for
20:
21:         return next_codes
22:     end if
23: end function
```

3.3 生成した攻撃コード集合の実行

次に，生成された攻撃コード集合に含まれるコードを実行し，それぞれのコードが送出するパケットをキャプチャし，一連の HTTP リクエストを取得する．Linux コンテナ化ソフトウェアである Docker を利用して，攻撃側とその攻撃対象となる多数の HTTP サーバをコンテナにより実装した．なお，本手法では HTTP サーバ上で攻撃コードの対象である特定のソフトウェアを稼働させていないため，サーバからの HTTP レスポンスにはシグネチャ作成のために必要な情報が含まれていないと考え，レスポンスのキャプチャは行なっていない．

また，Metasploit では，脆弱性を利用した攻撃後にリバースシェルの確立などを行うバイト列やコマンド列が `payload` モジュールとして存在し，攻撃コードの実行の際に複数の `payload` の中から任意に選択することができる．Metasploit による攻撃を検知する際に，そのような `payload` の特徴を検知するルールを定義するアプローチも考えることができるが，攻撃コードの特徴と異なり，`payload` は比較的容易に難読化可能なため，その特徴の検知は難しい．そこで，本研究では，攻撃コード実行の際に選択できる複数の `payload` のうち，有効な通信を行わないと判断した 3 つを除く全てを選択して実行したリクエスト列をキャプチャし，それらのリクエスト列に共通する特徴の抽出によって，生成された Snort ルール内に `payload` 由来の特徴が現れないようにした．

3.4 HTTP リクエスト列からの特徴抽出

3.4.1 概要

1 つの攻撃コードに対して異なる条件分岐を取ったり payload モジュールを選択したりして実行したとしても，それぞれに対して生成されたリクエスト列に対して先頭から順に集合を構成すると，それぞれの集合には似た傾向を示すリクエストが含まれていると考えられる．例えば，攻撃コード集合に含まれる攻撃コードを実行した結果，リクエスト列として $(h_{11}, h_{12}), (h_{21})$ が得られた際に，$\{h_{11}, h_{21}\}, \{h_{12}\}$ を構成する．このとき，それぞれの集合は，対象の脆弱性の存在を確認するリクエスト，実際に攻撃を行うリクエストを多く含むという状況が考えられる．そのような集合毎に特徴的な文字列の抽出を行うことで，その集合内のリクエストを検知するルールの生成を目指す．なお，この手法によって生成されるルールは，content オプションの値に抽出された文字列を指定することを想定している．

本手法の入力は，上で説明したような HTTP リクエストの集合 H で，出力は content で指定する文字列の集合から成る集合 R である．例えば，$R = \{\{$"POST","/vuln.php","exploit"$\}\}$ が得られた場合に生成されるルール例を**図 3** に示す．具体的な手続きとしては，まず，H 内のリクエストをパースし，その結果得られるメソッド名などの文字列毎に集合を計算した E_{all} を得る．例えばパースによって HTTP リクエストのメソッド，URL，リクエストボディに関する集合が得られた場合，$E_{all} = \{E_{method}, E_{path}, E_{body}\}$ であり，また，$E_{method} = \{$"POST","GET"$\}$ などの値が想定される．次に，n-gram による文字の出現確率のモデリングを用いて，$E \in E_{all}$ に対して，ルールの候補となる文字列集合 S_E の生成を行う．その後，S_E を元に，3.4.3 項に示すアルゴリズムによって，各 E に対して Snort ルールで指定する文字列の集合から成る集合 R_E を求め，R を計算する．

次に，n-gram による S_E の計算と，S_E から R_E と R を計算する部分について詳説する．

3.4.2 n-gram による候補文字列集合 S_E の生成

HTTP リクエストの集合から特徴となる文字列を抽出するために，まずそれぞれのリクエストをパースすることでメソッド，パス，各ヘッダ，本文の要素に分割する．その後，それぞれの要素について，文字ベースの n-gram によって文字の出現確率のモデリングを行なう．モデリングした結果を用いて，要素中に登場した文字 w_0 が与えられた際の文字列 $w_1 ... w_s$ の生成確率を (1) のように近似して計算する．

$$P(w_1 ... w_s | w_0) \simeq P(w_1|w_0) P(w_2|w_0 w_1) P(w_n|w_0 ... w_{n-1}) \prod_{i=n+1}^{s} P(w_i | w_{i-n} ... w_{i-1}) \quad (1)$$

次に，ルールの候補となる文字列の集合として，(1) で定義した確率が閾値 α を超えるような文字列の集合 S を，要素中に登場する文字 w_0 を始点とする枝刈り探索によって生成する．探索の際には，生成する文字列の最小長として m を設定し，短すぎる文字列は候補として生成されないようにした．$w_0 =$ "&", $\alpha = 0.25$，$m = 2$ とした場合の実行例を**図 4** に示す．この例では，$S = \{$"&id"$\}$ が得られた．

さらに，このアルゴリズムによって生成される文字列には，本来の HTTP リクエストの要素中には登場しないものも生成される可能性もあるので，最後にそのような文字列の除外を行う．除外の基準としては，頻度による手法を採用し，文字列 s が文字列 t の部分文字列であることを $s \subset t$ とすると，閾値 β を設定して，文字列 s について式 (2) が成り立つ場合に除外した．

$$\frac{|\{e \mid e \in E, s \subset e\}|}{|E|} < \beta \quad (2)$$

3.4.3 候補文字列集合 S_E を利用した R_E，R の生成

まず，前処理として，$s_1, s_2 \in S_E$ について，それらを結合した文字列 s が実際のリクエストを表す文字列内に含まれていた場合に，s を S_E に加える．例えば，$s_1 =$ "submit.php?name"，$s_2 =$ "name = Alice" であるとき，$s =$ submit.php?name = Alice とする．この操作を，このような関係を満たす s_1, s_2 が存在する限り行う．

続いて，$s_1, s_2 \in S_E$ について，$s_1 \subset s_2$ となる場合に，s_1 を除外する．この操作についても，$s_1 \subset s2$ なる $s1, s2$ が存在する限り実行する．以上の操作を前処理として実行

```
alert tcp any any -> any 80 (
    content:"POST";
    content:"/vuln.php";
    content:"exploit";
    sid: <ランダムな整数>;)
```

図 3 提案手法により生成される Snort ルールの例
Fig. 3 Example of generated Snort rule by proposed method

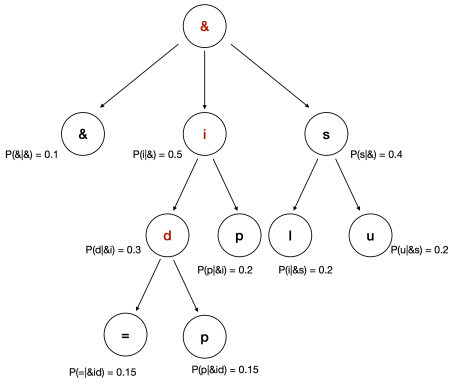

図 4 $\alpha = 0.25, m = 2$ とした場合の実行例
Fig. 4 Execution example given $\alpha = 0.25, m = 2$

することによって，より長い文字列が Snort のルールとして採用されるようにする．

次に，各 E について，content で指定する文字列の集合から成る集合 R_E を求める．これは，S_E の冪集合を $\mathfrak{P}(S_E)$ として，式 (3) によって計算する．

$$R_E = \{S \mid S \in \mathfrak{P}(S_E), \exists e \in E, \forall s \in S, s \subset e\} \quad (3)$$

これにより，S_E に含まれる全ての文字列の組み合わせの中から，実際に攻撃時の HTTP リクエストに登場した組み合わせを抽出する．また，この操作ののち，$r1, r2 \in R_E$ について，$r1 \subset r2$ なる関係を満たすものについて，$r1$ の除外を行う．包含関係にあるものの除外を行うことによって，例えば，攻撃対象のシステムの URL だけではなく，攻撃時にその URL で指定した対象に対して送出されるリクエストパラメータも合わせた Snort のルールが生成されるようにしている．

最後に，各 E から生成された R_E から式 (4) を用いて計算することによって，HTTP リクエストの集合 H を検知するような Snort ルールの content オプションに指定する文字列の集合からなる集合 R を計算する．なお，式中に現れるリクエスト h は文字列として扱う．

$$R = \{S \mid S \in \mathfrak{P}\left(\bigcup_{E \in E_{all}} R_E\right), \exists h \in H, \forall s \in \bigcup S, s \subset h\} \quad (4)$$

4. 実験

本章では，提案手法を実装したシステムを実行し，提案手法が適用できる攻撃コードなどについて評価する．実験は Intel Xeon 2.6GHz CPU と 32GB RAM を搭載したマシン上で行った．また，利用した主要なソフトウェアは，Ubuntu 18.04.3 LTS，Ruby 2.6.5，Python 3.6.9, Metasploit 5.0 である．攻撃コードの実行の際には，攻撃側とその攻撃対象となるサーバを 100 ペア用意し，並列して実行した．

4.1 生成される攻撃コードの総数

本節では，Metasploit の攻撃コードのうち，HTTP によるサーバへの攻撃を行うものについて，その制御パスの本数の評価を行なった．

手法 2020/08/04 時点で Metasploit に存在する攻撃コード[*5]のうち，HTTP によるサーバへの攻撃に利用されるモジュール HTTPClient[*6] を利用しているコード 718 個について，抽象構文木を用いた提案手法によって列挙される制御パスの本数の評価を行なった．実際には列挙の後，それぞれのパスと一対一に対応した攻撃コードにより集合を構成するため，制御パスの本数がそのコード集合のサイズに一致する．

結果 制御パスの本数についてまとめたものを表 1、表 2 に示す．攻撃コードに存在する制御パスの本数は指数的に増加しているが，一部を除いて一定の範囲に収まっていることが分かる．生成した攻撃コード集合を実行するための現実的な実行時間を考慮して，本数が 100000 以下ならば本手法が適用可能だとすると，83% のコードが該当した．また，該当コードについて，それぞれに対するコード集合の生成時間を計測すると，その平均は 2.22 秒で標準偏差は 10.1 秒であった．

表 1 制御パス数の統計量
Table 1 Number of control paths summary

平均値	中央値	最小値	最大値
9.22×10^{14}	64	1	6.49×10^{17}

表 2 制御パス数の x% 点
Table 2 x percentile

x	0%	25%	50%	75%	100%
制御パス数	1	4	64	1536	6.49×10^{17}

4.2 事例紹介

本節では，いくつかの攻撃コードについて，提案手法によって出力された Snort のルールを示す．表 3 に，生成の際に指定したパラメータについて示す．

4.2.1 ShellShock

ShellShock とは，2014 年 9 月に公開された Bash における任意コマンド実行可能性に関する脆弱性であり，環境変数に特定の文字列とそれに続くコマンド列の代入を試みることで攻撃を行う．Metasploit においては，環境変数を利用するソフトウェア経由でこの攻撃を実行するようなコードが，それぞれのソフトウェアに対する攻撃コードとして用意されているが，本稿では，その中でも Apache Web サーバに対して不正なリクエストを送出することで攻撃を実行するモジュール，apache_mod_cgi_bash_env_exec[*7]について説明する．

この攻撃コードによる攻撃では，**図 5** のような関数によって生成された文字列を User-Agent の値として利用する．そのため，この攻撃コードによる攻撃に対する Snort の

表 3 ルールの生成を行う際に設定したパラメータ
Table 3 parameters

パラメータ	n	α	β	m
値	4	0.18	0.3	10

[*5] https://github.com/rapid7/metasploit-framework/tree/master/modules/exploits
[*6] https://www.rubydoc.info/github/rapid7/metasploit-framework/Msf/Exploit/Remote/HttpClient
[*7] https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/multi/http/apache_mod_cgi_bash_env_exec.rb

ルールとしては，“() { :;};”，“() { _; } >_[$($())]”のような文字列を指定することが考えられる．実際，Snort の Community 版ルール[*8]では，ShellShock に対する攻撃を検知するルールとして，“() {”のような文字列の指定によるものが多数存在している．

図 6 に，提案手法によって生成されたルール集合を示す．前半の 2 つが実際に攻撃を行なっている HTTP リクエストの集合から生成されたルール集合，後半の 2 つがサーバに対して ShellShock が実行できるかを確認するリクエストの集合から生成されたルール集合である．前半の 2 つでは，それぞれ 1 つ目と 3 つ目の `content` に `payload` モジュール由来の特徴が抽出されているが，これらは特定の `payload` に依存しないものであるため，有効なルールが生成できていると考える．また，生成されたルール集合は Community 版で定義されているルールとは完全に一致しないが，Metasploit による ShellShock への攻撃の検知に対しては，必要なルールの生成ができていると考える．

この攻撃コードからはサイズ 96 の攻撃コード集合が生成され，それぞれの実行時に 27 の `payload` が選択された．そして，全ての実行に 9 分 39 秒，ルールの生成に 1 分 8 秒を要した．

4.2.2 不正なファイルアップロードとその実行に関する脆弱性

不正なファイルアップロードに関する脆弱性を攻撃するモジュール，`apprain_upload_exec`[*9] に対して提案手法を適用した．このモジュールでは，次のような流れで攻撃対象のサーバ上における任意コマンドの実行を行う．

(1) `uploads` ディレクトリに任意コマンドをを記述した PHP ファイルをランダムな名前でアップロード
(2) アップロードした PHP ファイルに外部からアクセスすることで，そのコマンドを実行する

ここで，**図 7** に提案手法によって生成されたルール集合の一部を示す．これはアップロードした PHP ファイルを実行する HTTP リクエストの集合に対して生成されたものであると推測されるが，アップロードされたファイル名はランダムに決定されるため，提案手法では抽出されなかった．しかし，このルールでは通常のアクセスに対しても反応してしまい，偽陽性が高くなることが問題となると考えられる．このような問題は，不正なファイルアップロードとその実行のような，複数のリクエストを経て実行される攻撃に対して提案手法を適用した場合に見られた．

また，この攻撃コードから生成された攻撃コード集合のサイズは 16 であった．そして，それぞれの実行時に 43 の `payload` が選択され，全ての実行とルールの生成にそれぞれ 6 分 32 秒，24 秒を要した．

```
def cve_2014_6271(cmd)
    %Q{() { :;};
        echo −e "\\r\\n#{marker}$(#{cmd})#{marker}"}
end
def cve_2014_6278(cmd)
    %Q{() { _; } >_[$($())]
        { echo −e "\\r\\n#{marker}$(#{cmd})#{marker}"; }}
end
```

図 5 ShellShock のトリガーとなる文字列を生成する関数
Fig. 5 Functions to generate a string that triggers ShellShock

```
alert tcp any any −> any 80 (
    content:"$(echo −en \\\\x";
    content:"_\; } >_[$($())] { echo −e \"\\r\\n";
    content:"\; /bin/chmod 777 /tmp/";
    content:"application/x−www−form−urlencoded";
    sid:156283; )
alert tcp any any −> any 80 (
    content:"$(echo −en \\\\x";
    content:"{ :\;}\;echo −e \"\\r\\n";
    content:"\; /bin/chmod 777 /tmp/";
    content:"application/x−www−form−urlencoded";
    sid:140014; )
alert tcp any any −> any 80 (
    content:"application/x−www−form−urlencoded";
    content:"_\; } >_[$($())] { echo −e \"\\r\\n";
    sid:188352; )
alert tcp any any −> any 80 (
    content:"application/x−www−form−urlencoded";
    content:"{ :\;}\;echo −e \"\\r\\n";
    sid:100918; )
```

図 6 実際に生成されたルール集合
Fig. 6 An example of the generated Snort rule set

```
alert tcp any any −> any 80 (
    content:"application/x−www−form−urlencoded";
    content:"/addons/uploadify/uploads/";
    content:"Mozilla/4.0 (compatible\; MSIE 6.0\;
        Windows NT 5.1)";
    sid:158315; )
```

図 7 `apprain_upload_exec` に対して生成されたルールの一部
Fig. 7 Subset of generated rule set for `apprain_upload_exec`

5. 考察

5.1 攻撃コード集合のサイズとその生成時間に関する問題

一般に，プログラム中における制御パスの本数は，if 文などの制御構造の個数を n として $\mathcal{O}(2^n)$ で増加する．そのため，制御構造を多く含む攻撃コードに対して提案手法を適用すると，現実的な時間内で計算が終了しなくなってしまう．実際，4.1 節で示した通り，一部のコードについては，組合せ爆発によって攻撃コード集合の生成が不可能であることが示された．しかし，その割合については決して高くないため，実行時間の面では，多くの攻撃コードに

[*8] https://www.snort.org/downloads
[*9] https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/multi/http/apprain_upload_exec.rb

ついて提案手法によるルール生成が利用できると考える．

5.2 実装・メンテナンスのコストの評価

提案手法を Metasploit 以外のツールにおける攻撃コード，また，一般的な PoC コードに対して適用するためには，そのコードをパースした結果の抽象構文木にアクセスできる，安定したインターフェースが必要である．例えば，PoC コードの開発で利用されることの多い Python では，そうしたインターフェースが標準ライブラリとして提供されている[*10]．そのため，Python によるコードに対して本手法を適用するためのハードルは比較的低いと言える．しかし，このような抽象構文木へのインターフェースが提供されていない場合は、提案手法を実装することは難しい．

また，メンテナンスのコストを考える際に問題となるのは、言語やツールの仕様追加・変更である。例えば、Ruby2.7.0 では新しくパターンマッチが導入されたが[*11]、変数の展開とその評価による条件分岐が必要なパターンマッチに対して，提案手法のような単純な静的解析によってそれぞれの制御パスに従ったコード集合を生成することは難しい．このように，提案手法は言語の仕様変更によってメンテナンスコストが増大する可能性がある．

5.3 生成される Snort ルールの特徴

4.2.1 節に示した通り，ShellShock に対する攻撃コードに対して本手法を適用することによって，その実行時のリクエストに特徴的な文字列を抽出し，Snort ルールの生成を行うことができた．そのため，1 つのリクエストによる攻撃が行われ，またそのリクエスト中に特徴的な文字列が含まれる攻撃については，提案手法を適用することによって同様にルールを生成できる可能性がある．

しかし，4.2.2 節で述べた通り，複数の HTTP リクエストによる攻撃に対して提案手法によって生成される Snort ルールは，正常な通信時にも見られる特徴のみを含んでいるため，偽陽性が高いと考えられるものが生成される可能性がある．これは，提案手法が攻撃コード集合の実行時に得られたリクエストから抽出した特徴が，全て Snort のルールとして有用なものであると仮定していることに起因する．そのため，このような偽陽性が高いと考えられるルールの生成を防ぐには，例えば正常な通信時の HTTP リクエストの集合を別に用意し，そこから抽出した特徴と比較することで，攻撃時特有の特徴のみが利用できるようにするなどの工夫が考えられる．

6. おわりに

本稿では，Metasploit の攻撃コードのうち，HTTP によるサーバへの攻撃を行うものについて，その攻撃コードから自動で Snort のルールを生成する手法について提案した．攻撃コードの抽象構文木を利用した制御パスの全列挙と，それぞれのパスに従った攻撃コードを実行することによって，シンボリック実行の全パス網羅性を模倣しつつ，実装・メンテナンスコストを下げることを目指している．また，特定の脆弱性に対して，提案手法によって生成されたルール例を示し，提案手法が ShellShock や類似した脆弱性に関しては適用可能である可能性を示した．今後は，今回提案した Snort のルール生成手法を用いて，生成されたルールによる検知の精度や偽陽性の存在について定量的な評価を行う．

[*10] https://docs.python.org/ja/3/library/ast.html
[*11] https://www.ruby-lang.org/ja/news/2019/12/25/ruby-2-7-0-released/

参考文献

[1] Baah, G. K., Hobson, T., Okhravi, H., Roberts, S. C., Streilein, W. W. and Yuditskaya, S. C.: A Study of Gaps in Cyber Defense Automation, Technical report, Defense Technical Information Center (2016).

[2] Kim, H.-A. and Karp, B.: Autograph: Toward Automated, Distributed Worm Signature Detection, *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, USA, USENIX Association, p. 19 (2004).

[3] King, J. C.: Symbolic Execution and Program Testing, *Commun. ACM*, Vol. 19, No. 7, p. 385394 (1976).

[4] Kreibich, C. and Crowcroft, J.: Honeycomb: Creating Intrusion Detection Signatures Using Honeypots, *SIGCOMM Comput. Commun. Rev.*, Vol. 34, No. 1, p. 5156 (2004).

[5] Moser, A., Kruegel, C. and Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis, *2007 IEEE Symposium on Security and Privacy (SP '07)*, pp. 231–245 (online), DOI: 10.1109/SP.2007.17 (2007).

[6] Newsome, J., Karp, B. and Song, D.: Polygraph: automatically generating signatures for polymorphic worms, *2005 IEEE Symposium on Security and Privacy (S P'05)*, pp. 226–241 (2005).

[7] Ramirez-Silva, E. and Dacier, M.: Empirical Study of the Impact of Metasploit-Related Attacks in 4 Years of Attack Traces, *Advances in Computer Science – ASIAN 2007. Computer and Network Security* (Cervesato, I., ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 198–211 (2007).

[8] Singh, S., Estan, C., Varghese, G. and Savage, S.: The EarlyBird system for Real-time Detection of Unknown Worms (2003).

[9] Wang, R., Ning, P., Xie, T. and Chen, Q.: MetaSymploit: Day-One Defense against Script-based Attacks with Security-Enhanced Symbolic Analysis, *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C., USENIX, pp. 65–80 (2013).

[10] You, I. and Yim, K.: Malware Obfuscation Techniques: A Brief Survey, pp. 297–300 (online), DOI: 10.1109/B-WCCA.2010.85 (2010).

[11] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao and Chavez, B.: Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience, *2006 IEEE Symposium on Security and Privacy (S P'06)*, pp. 15 pp.–47 (2006).