Webアプリケーションを安全にするフレームワークの新しい機能

久保田 康平

令和　3年　2月

情報知能工学専攻

概 要

本論文は，Webアプリケーションのセキュリティ機能向上を目的にしている．そのために本論文では，Webアプリケーション開発者が実装するコードを実行時に自動的に解析し，必要ならば修正する機能をWebアプリケーションフレームワークに持たせることを提案し，実装して評価を行う．Webアプリケーションはインターネットを通して世界中から誰でも接続でき，対話的に通信できるという特徴から様々な攻撃の対象になる．また，インターネットの普及に伴いWebアプリケーションの重要性は増し，同様にWebアプリケーションの防御もまた重要になっている．脆弱性攻撃は，Webアプリケーションの設計上の欠点や仕様上の問題点である脆弱性を利用する攻撃である．脆弱性攻撃の対策の1つは，Webアプリケーションに脆弱性を作らないことであり，そのためWebアプリケーション開発者は一般的にWebアプリケーションフレームワーク[?][?][?]を利用する．Webアプリケーションフレームワークは，Webアプリケーション開発において利用することが多いメソッドを持つライブラリである．それらのメソッドを利用

することで効率よく安全なアプリケーションを開発することができる．セキュリティ面において，Web アプリケーションフレームワークが提供するメソッドは脆弱性対策がなされているものが多い．したがって，Web アプリケーションフレームワークを利用した方が，利用しない時と比較して効率的にセキュアな Web アプリケーションを開発しやすい．一方で，開発者は常に完全にセキュアなコードを書くことはできないため，Web アプリケーションフレームワークを利用して，脆弱性がある Web アプリケーションを実装してしまうことがある．その理由の 1 つが，Web アプリケーション開発者が Web アプリケーションフレームワークを適切に利用できないことである．Web アプリケーション開発者が，フレームワークのメソッドが持つセキュリティ機能を正しく理解していなかったり，セキュリティ機能を持つメソッドを知らなかったりすることによって脆弱な Web アプリケーションが実装される．この問題に対して本論文では，Web アプリケーション開発者が実装したソースコードを修正する機能を持つ Web アプリケーションフレームワークを提案する．提案手法を実証し評価を行った結果，この機能は実装されたコードの脆弱性を一部修正でき，レスポンスタイムは提案手法を適用しなかった場合とほとんど変わらないことを確認した．実装された修正関数の蓄積は将来のアプリケーションのセキュリティの向上に寄与できるものである．

目 次

図 目 次

第1章

はじめに

本論文は，Web アプリケーションのセキュリティ機能向上を目的にしている．その目的の達成のために，アプリケーション開発者が実装したプログラム中の関数や引数を解析し，実行時にその関数に脆弱性があった時には修正することができる Web アプリケーションフレームワークを提案，実装し評価する．

Web アプリケーションセキュリティは，セキュリティ分野において重要である．インターネットの普及に伴い，Web システムは様々な場所や階層において様々な攻撃にさらされている．Web システムへの攻撃のうちアプリケーション層への攻撃の多くはアプリケーションのプログラムが持つ論理的な問題が原因である．そのため Web アプリケーション開発者は攻撃を回避するために，アプリケーションの論理的な問題や設計上の欠点である脆弱性を作らない実装をする必要がある．一方で，Web アプリケーション開発者は常にセキュアなコードを記述することはできず，脆弱性を残す実装をすることがある．加えて Web アプリケーション層にはセ

キュリティに関するプロトコルや標準的な仕様がないため，Web アプリケーションの安全性は，Web アプリケーション開発者のセキュリティに関する知識や技術に依存する．これらの Web アプリケーションの問題を解決しセキュリティを向上するために，Web アプリケーションの自動防御手法として Web アプリケーションファイアウォール [?][?]（WAF）や Web アプリケーションフレームワーク [?][?][?] の利用などが検討されている．

WAF は，Web アプリケーションを脆弱性攻撃から保護するためのシステムである．WAF は Web アプリケーションとクライアントの間に配置され，クライアントからのリクエストを監視し，リクエストが攻撃リクエストかどうかを検証する機能を持つ．攻撃を検出した場合，そのリクエストを遮断もしくは無毒化することで，Web アプリケーションへの攻撃の影響を低減する．WAF は Web アプリケーションを修正することなく，脆弱性攻撃を低減することが可能であるため，アプリケーションを直接修正できない時に有効な対策である．一方で WAF はアプリケーションを修正しないので，アプリケーション内の脆弱性を根本的に修正できないという欠点がある．また WAF はアプリケーション内の論理的な設計や仕様を知らないため，一部のタイプの脆弱性を対策することが難しい．WAF は通常，特殊文字を含むリクエストを攻撃として検出する．したがって，リクエスト内に特殊文字を含まない攻撃を WAF が検出することは難しい．

Web アプリケーションフレームワークは，Web アプリケーションを効率よく開発するために，Web 開発に多用される機能を関数やメソッドとして提供するライブラリである．自動防御手法としては，クロスサイトスクリプティング（XSS）や SQL インジェクション（SQLi）のようなイ

ンジェクション攻撃に対する入力検証と自動サニタイズという機能を提供していることがある．自動サニタイズとは特殊文字をエスケープする機能であるサニタイズを Web アプリケーションフレームワークが行う一部の Web アプリケーションフレームワークが持つ機能である．自動サニタイズの長所は Web アプリケーションのセキュリティの一部を Web アプリケーションフレームワークが負担することが可能なことである．自動サニタイズによって Web アプリケーション開発者はサニタイズについて考慮することなく，セキュアな Web アプリケーションを実装することが可能になる．一方で自動サニタイズは限定的な対策で，インジェクション攻撃ではない攻撃を対策することが難しい．

Web アプリケーションの自動防御は Web アプリケーションの論理的な設計を検証し脆弱性の影響を低減する機能を持たないため，一部の攻撃を自動的に防御することができない．具体的には，Web アプリケーションの不適切な認証への攻撃を自動で対策する手法を Web アプリケーションフレームワークは持たない．不適切な認証は，アプリケーションの利用者が権限を所持していると主張した時に，アプリケーションがその主張が適切かどうかを証明しない，もしくは不適切に証明する脆弱性である．

この問題を解決するために，本論文ではアプリケーション開発者が実装したソースコードを解析し，必要であれば修正する Web アプリケーションフレームワークである VHF（Vulnerability Handling Framework）を提案する．VHF は脆弱なソースコードの条件とそのソースコードを修正するプログラムを持っている．Web アプリケーション開発者が記述したソースコードを実行開始時に静的に解析することで脆弱性を検出する．その後，脆弱なソースコードを保護するための関数を挿入したり，安全な関

数に置き換える．挿入された関数は実行中にアプリケーションを動的に検証し，攻撃を検出すると無毒化する．この提案手法の貢献は，Web アプリケーション開発者のソースコードを自動で修正するので，そのアプリケーションの論理的設計の不備を修正することが可能であることである．したがって VHF は通常の Web アプリケーションフレームワークでは対策できない認証の不備に対する攻撃の低減を行うことが可能である．

VHF は実行開始時にアプリケーション開発者が実装したソースコードを修正するシステムとリクエストを処理するシステムの 2 つで構成されている．ソースコードを修正するシステムは Web アプリケーション開発者が実装したソースコードを解析し修正する機能である．VHF は実行開始時にアプリケーション開発者が実装したソースコードをフレームワーク内に格納し，その後格納したソースコードを解析・修正する．リクエストを処理するシステムはクライアントからリクエストを受け取りレスポンスを作成して応答するシステムである．具体的にはまず受け取ったリクエストをアプリケーションが処理しやすい形式に変更する．次にそのリクエストを用いてレスポンスボディを作成する．最後にレスポンスを作成する．リクエストに基づいてレスポンスヘッダーを作成し，レスポンスボディと組み合わせてレスポンスを作成する．作成されたレスポンスはクライアントに返される．

本論文では Web アプリケーション開発者が実装したソースコードを解析して脆弱性の影響を緩和することが可能であるかを確認するために実験を行った．その結果，不適切な認証と SQLi の脆弱性を修正できることを確認した．

第2章

関連研究

2.1　論文1

2.2　論文2

2.3　論文3

第3章

提案手法

この章では，コールバック関数を解析し必要であれば修正する Web アプリケーションフレームワークである VHF を提案する．コールバック関数とは Web アプリケーションへのリクエストを基に，Web サーバー側で行う処理を記述した関数である．図 3.1 は VHF の概要図である．VHF は

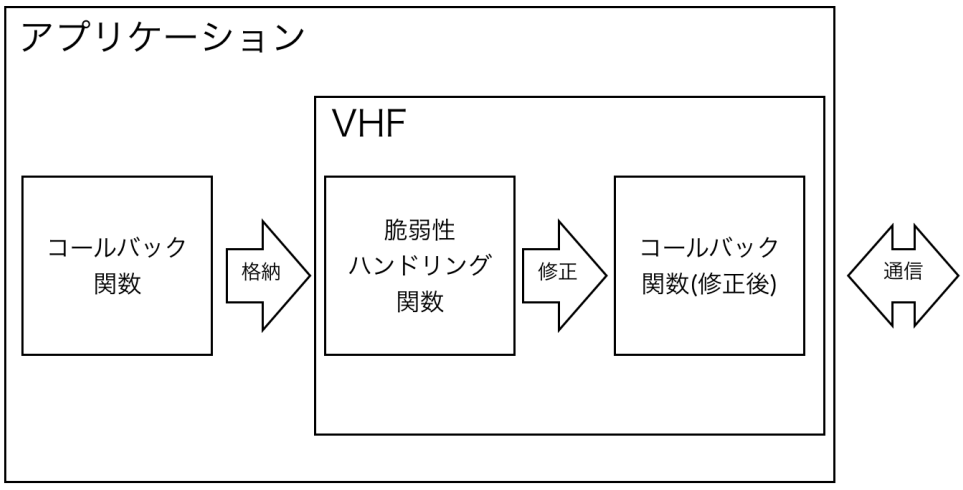

図 3.1: VHF の概要図

コールバック関数の修正機能とリクエストの処理機能を持っている．VHF が実行されるとまずコールバック関数を修正する．具体的には実行開始時に VHF はコールバック関数を VHF 内に格納し，コールバック関数を修正する．その後実行中はクライアントからのリクエストに対して修正されたコールバック関数で処理を行うことで脆弱性の影響を低減する．

VHF はサイバーセキュリティの観点から 3 つの利点がある．まず 1 つ目に VHF が対策しているコールバック関数に関するセキュリティ機能の全てをアプリケーション開発者が負担しなくていい点である．一般的な Web アプリケーションフレームワークにおける Web アプリケーションの実装では，Web アプリケーション開発者は安全なコールバック関数を実装する全責任を負っている．VHF はコールバック関数を自動で解析・修正するので，VHF がコールバック関数の責任の一部を担うことが可能である．第 2 に，VHF は WAF で対策が難しい脆弱性攻撃を対策可能である．一般的な WAF はアプリケーションへのリクエストを解析することで脆弱性攻撃の影響を低減する．具体的には，リクエスト中の特殊文字や他のプログラミング言語に関する意味のある文字を脆弱性攻撃としそれらを検出する．この検出手法は Web アプリケーションを修正することなく Web アプリケーションを自動で防御できるが，リクエスト中に特殊文字を含まない攻撃を対策することが困難である．リクエスト中に特殊文字を含まない攻撃の 1 つが不適切な認証を持つアプリケーションへの攻撃である．不適切な認証は，Web アプリケーションの論理的な設計の間違いが原因で起こる脆弱性である．不適切な認証によって正しいアクセス制御ができず，それにより特権を持たないユーザーが特権リソースに接続する攻撃が発生する．VHF はコールバック関数間の論理的な設計を

比較して他のコールバック関数に適用することで不適切な認証の影響を低減することが可能である．第 3 に，VHF はコールバック関数を実行開始時に修正するので脆弱性を根本的に対策することが可能である．WAF は，コールバック関数の脆弱性を削除できないため，特定の攻撃を対策してもその対策を迂回する攻撃が発生する可能性がある．

VHF は図 3.2 に示す通り，実行開始時に 4 つの工程によってコールバック関数を解析・修正する．図 3.2 の上部はコールバック関数の形式であり，下部が VHF 内部で行われる工程である．まず最初に，コールバック関数，

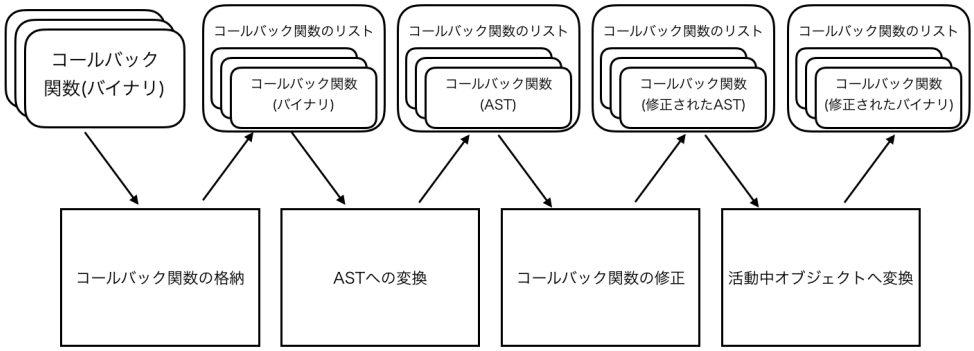

図 3.2: コールバック関数を修正するための 4 つの工程

リクエストパス，メソッドを 1 つの辞書式データとして VHF に格納する．この辞書式データはリストの一要素として VHF に格納される．この時点ではコールバック関数は活動中のオブジェクト，つまり実行可能なバイナリ形式のオブジェクトである．次にコールバック関数を修正しやすくするために，VHF はコールバック関数の形式を活動中のオブジェクトから抽象構文木 [?]（Abstract Syntax Tree: AST）に変更する．AST は，プログラムを実行可能なバイナリ状態にする処理の途中で取得される中間

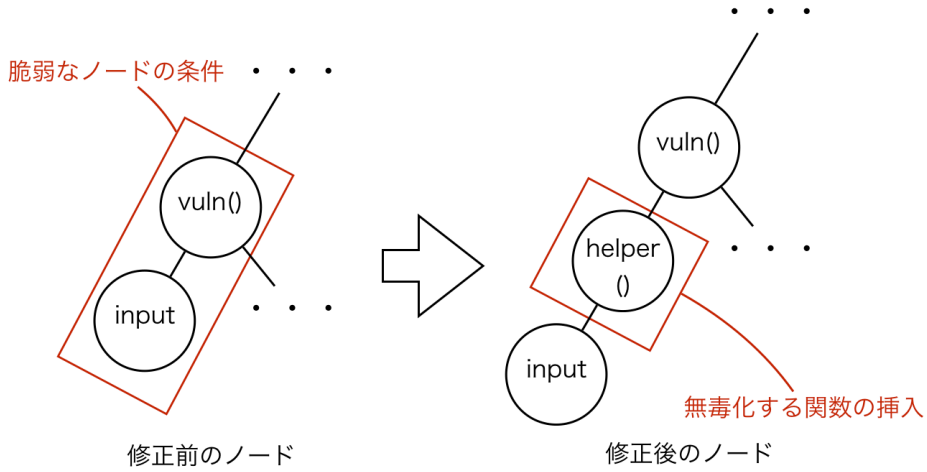

図 3.3: AST の修正による脆弱性影響低減手法の概要図

生成物であり，ソースコードから実行可能なオブジェクトを生成するために必要ない部分を削除した表現である．AST はバイナリよりもプログラムの論理的設計を把握しやすいため，コールバック関数の解析と修正が容易である．3 つ目がコールバック関数の解析と修正である．VHF はコールバック関数を解析し修正する脆弱性ハンドリング関数を持っている．脆弱性ハンドリング関数は特定の属性と名前を持つノードを脆弱なノードとし，このノードを再帰的に探索して検出して，その後脆弱性ハンドリング関数の処理によってノードの一部を修正される．具体的な修正方法は，脆弱なノードに VHF が持つ関数やメソッドを挿入する．脆弱性ハンドリング関数は図 3.3 に示すように，脆弱なノードの一部に攻撃を無毒化する関数を挿入することで，実行時に変数を動的に検証し，攻撃を無毒化する．図 3.3 では，vuln() 関数を脆弱なノードの条件としており，vuln() 関数の引数 input を無毒化する関数 helper() を挿入することで，脆

弱性の影響を低減している．脆弱性ハンドリング関数はコールバック関数のリストを引数として受け取る．このリストは全てのコールバック関数が AST の状態で格納されている．脆弱性ハンドリング関数は全てのコールバック関数を受け取ることで，単一のコールバック関数内の脆弱性だけでなく，コールバック関数間の論理的な設計によって起きる脆弱性を対策することが可能である．脆弱性ハンドリング関数はコールバック関数を修正したのち，全てのコールバック関数が格納されたリストを返す．最後に，修正されたコールバック関数の AST を実行可能なオブジェクトに戻す．実行可能な状態になった修正されたコールバック関数は，クライアントの通信の際に呼び出されリクエストを処理することができる．

第4章

実装

この章では，VHF の実装について記述する．VHF は Python3.7 によって実装されている．VHF は 2 つのシステムによって構成されている．コールバック関数を修正するシステムとリクエストを処理するシステムである．実行開始時にコールバック関数を修正するシステムにより，コールバック関数が修正され，実行中はクライアントのリクエストに基づきレスポンスを返す．

4.1　コールバック関数の修正機能

VHF は実行開始時にコールバック関数を解析･修正する．この時にコールバック関数の格納，コールバック関数の AST への変換，コールバック関数の修正，コールバック関数の活動中のオブジェクトへの変換が行われる．それぞれの実装について下に記述する．

4.1.1　コールバック関数の格納

VHF はコールバック関数を格納するためにデコレータを利用するためのメソッドとして route メソッドを持つ．実効開始時に route メソッドはコールバック関数を VHF 内のリストに格納する．以下のソースコードはコールバック関数の例である．

```
1 @app.route(path='^/$', method='GET')
2 def index(request):
3   return "Hello"
```

Listing 4.1: コールバック関数の一例

ソースコード 4.1 の 1 行目がデコレータである．デコレータは関数を修飾する関数であり，下記のソースコード 4.2 はソースコード 4.1 と糖衣構文である．デコレータを利用することで関数を引数にする関数の記述を簡易にしてくれる．

```
1 def index(request):
2   return "Hello"
3 index = app.route(path="/", method="GET")(index)
```

Listing 4.2: ソースコード 4.1 と糖衣な表現

ソースコード 4.1 の 1 行目にある app は VHF のモジュールであり，route は app モジュールが持つメソッドの 1 つである．route メソッドはリクエストパスとリクエストメソッドを引数としている．ソースコード 4.1 の path がリクエストパスの正規表現，method がリクエストメソッド，2 行目と 3 行目の関数が第 3 引数のコールバック関数である．コールバック

関数は request を引数として受け取る．request はリクエストの情報を格納している変数である．ソースコード 4.1 の 3 行目は戻り値であり，この戻り値はその後レスポンスボディになる．route メソッドはリクエストパスとリクエストメソッドをコールバック関数と対応付けて VHF に格納する．以下のソースコード 4.3 が route メソッドの実装である．

```
1 class App():
2   ...
3     def route(self, path=None, method='GET'):
4         def decorator(callback_func):
5             self.router.add(method, path,
                callback_func)
6             return callback_func
7         return decorator
```

Listing 4.3: route メソッド

route メソッドを実行すると route メソッド内部の decorator 関数を返す．その後コールバック関数を引数とした decorator 関数が実行される．decorator 関数内の router.add メソッドはコールバック関数を VHF に格納するメソッドである．decorator 関数はコールバック関数とリクエストパス，リクエストメソッドの要素を持つ辞書形式にして，その辞書データをリストに格納する．

4.1.2　コールバック関数の AST への変換

VHF に格納された時点ではコールバック関数は活動中のオブジェクト，つまり機械語である．機械語を解析・修正するのは容易ではないため格納されたコードを一度修正しやすい形式に変換する．具体的には，活動中のオブジェクトを AST に変換する．Python は活動中のオブジェクトを AST に直接変換できないため，活動中のオブジェクトを一度ソースコードに変換したのちに AST に変換する．活動中のオブジェクトからソースコードを取得するために，Python が提供している inspect モジュールの getsource メソッドによりソースコードを取得する．getsource メソッドは活動中のオブジェクトを引数に取り，そのソースコードを返すメソッドである．Python の活動中オブジェクトは関数名やソースコードのファイルのパスなどの情報を保持している．その情報を利用して getsource メソッドはソースコードのファイルを読み取り，ソースコードを取得する．動的に実行された活動中のオブジェクトには，ソースコードのファイルなどの情報がないため，getsource メソッドでソースコードを取得できない点には注意が必要である．getsource メソッドが取得したソースコードは route デコレータを含むため，そのまま AST に変換できない．そのため，コールバック関数の先頭にある route デコレータを取得したソースコードから取り除く．その後，取得したソースコードを AST に変換する．Python は，AST を処理するライブラリとして ast モジュールを提供している．このモジュールは，ソースコードを AST に変換したり AST を探索したりするメソッドを持っている．parse メソッドは ast モジュール内にある，ソースコードを AST に変換するメソッドである．parse メソッドはソースコー

ドを引数として受け取るため，格納された時点での活動中のオブジェクトとしてのコールバック関数を受け取ることはできない．したがって，一度コールバック関数をソースコードに変換したのち，parse メソッドを利用してコールバック関数をソースコードから AST に変換する．生成された AST は，リクエストメソッドとリクエストパス，AST を要素とする辞書形式にまとめられ，この辞書型式のデータはリストに格納される．

4.1.3　コールバック関数の修正

その後 VHF は AST になったコールバック関数を解析・修正する．VHF はコールバック関数を修正する関数として脆弱性ハンドリング関数を実装している．脆弱性ハンドリング関数は AST であるコールバック関数のリストを引数に取り，解析・修正された AST であるコールバック関数のリストを返す．脆弱性ハンドリング関数の内部では脆弱性と判断したノードがコールバック関数内にあるかを解析し，検出されたノードを脆弱性ハンドリング関数内で定義したノードの修正方法に基づいて修正する．

脆弱性ハンドリング関数は新しい脆弱性が発見された時に追加の実装をしやすくするために脆弱性ハンドリング関数を分割している．脆弱性ハンドリング関数は脆弱性ごとに分割されており，これらの分割された脆弱性ハンドリング関数はリストに格納される．コールバック関数のリストは図 4.1 に示すように順に脆弱性ハンドリング関数によって解析・修正される．

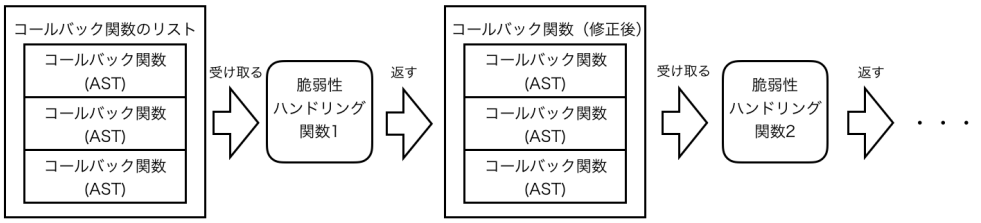

図 4.1: 脆弱性ハンドリング関数がコールバック関数を修正する概略図

4.1.4　コールバック関数の活動中のオブジェクトへの変換

最後にコールバック関数を活動中のオブジェクトに変更する．AST を実行可能な形式に変更するために，exec() 関数を利用する．これにより，コールバック関数は活動中のオブジェクトとなり，リクエストを処理することが可能になる．

4.2　リクエスト処理システム

リクエスト処理システムはリクエストを基にコールバック関数を呼び出しレスポンスを作成するシステムである．このシステムはリクエスト情報の取得，コールバック関数の呼び出し，リクエストの作成の 3 つの工程でリクエストを処理する．

4.2.1　リクエスト情報の取得

リクエスト情報の取得は，クライアントからの HTTP リクエストを取得し，アプリケーションが処理しやすいように加工する工程である．この

工程では request インスタンスが作成される具体的には，リクエストパラメータを辞書形式に変換したり，リクエストボディをしたりして，HTTP リクエストを元に request インスタンスを作成する．request インスタンスのインスタンス変数がリクエスト情報になる．例えば request.path がリクエストパス，request.method がリクエストのメソッド，request.query がリクエストのパラメータである．このインスタンス変数はコールバック関数に引数として与えられる．

4.2.2　コールバック関数の呼び出し

リクエストパスとリクエストメソッドに基づき，コールバック関数を呼び出しレスポンスボディを作成する．VHF に格納されている修正されたコールバック関数のリストから，リクエストパスとリクエストメソッドが一致するコールバック関数を探す．一致するコールバック関数が存在する場合，requst を引数に与えてコールバック関数を実行する．コールバック関数の戻り値がレスポンスボディに相当する．一方，一致するコールバック関数がない場合,「Not Found」をレスポンスボディにする．

4.2.3　レスポンスの作成

レスポンスボディをエンコードし，レスポンスヘッダーを作成したのち，レスポンスヘッダーとレスポンスボディを組み合わせレスポンスを作成する．一致するコールバック関数が存在しレスポンスボディが作成された時は，レスポンスヘッダーのステータスコードを 200，存在しない

時は 404 とする．レスポンスヘッダーとレスポンスボディを組み合わせクライアントに送信する．

第5章

実験

VHF について 2 つの実験を行った．1 つ目に脆弱性ハンドリング関数が脆弱性の影響を低減したかどうかの評価を行った．2 つ目に脆弱性ハンドリング関数が実装されたことによる実行開始時のオーバーヘッドの計測を行った．本実験は以下の環境で行われた．Mac OS X El Capitan 10.11.6, Intel Core i5(2.95GHz), メインメモリ 8GB であった.

5.1　脆弱性の影響低減評価

脆弱性ハンドリング関数を実装することにより脆弱なコールバック関数を修正し，脆弱性攻撃への影響を低減することが可能か評価した．本実験では 2 つの脆弱性を持つアプリケーションを 1 つ実装した．実装されたアプリケーションが持つ 2 つの脆弱性は SQLi と不適切な認証である．この脆弱性に対して，それぞれ脆弱性ハンドリング関数を実装した．その後，ローカル上でアプリケーションを実行し，攻撃することで脆弱性攻撃の影響を低減できたか評価した．以下には，それぞれの脆弱なコー

ルバック関数と脆弱性ハンドリング関数を記述する．

5.1.1　SQLi を持つコールバック関数の修正と攻撃

SQLi はリクエスト内の値を利用して直接クエリを作成することで起こる脆弱性である．リクエストに特殊文字を挿入することで，アプリケーション開発者が意図していない命令がデータベースで実行される．これにより，データベースが改ざんされたり不正に削除されたりする．本実験では，以下の図に示すように SQLi 脆弱性を持つアプリケーションが実装された．下記のソースコード 5.1 は SQLi 脆弱性を持つアプリケーションのコールバック関数の一部である．

```
1  @app.route("^/access$", "POST")
2  def access(request):
3    import sqlite3
4    conn = sqlite3.connect("test.sqlite3")
5    cur = conn.cursor()
6    action = request.forms.get('action')
7    name = request.forms.get('name')
8    password = request.forms.get('password')
9    query = '{action} * from user'.format(action=
       action)
10   if action='select':
11     query += " where name = '{name}' and password =
         'password'".format(name=name, password=
```

```
        password)
12      cur.execute(query)
13      data = cur.fetchone()
14      return tmpl("access.html", action='select',tel=
            data[2],mail_address=data[3])
15    else:
16      cur.execute(query)
17    return tmpl("access.html", action=action)
```

Listing 5.1: SQLi 脆弱性を持つコールバック関数

上記のソースコード 5.1 は，リクエストパラメータに基づいてデータベースを操作し，その結果をクライアントに返すコールバック関数である．このソースコードの 1 行目は，コールバック関数を格納するメソッドである．リクエストパスが”/access”でリクエストメソッドが POST の時，このコールバック関数が呼び出される．2 行目以降の access() 関数がコールバック関数である．ソースコード 5.1 の 3 行目から 5 行目でデータベースと接続する準備をしている．3 行目でリレーショナルデータベースとして sqlite3 をインポートしている．4 行目でデータベースに接続し，5 行目でカーソルを宣言している．その後ソースコード 5.1 の 6 行目から 8 行目では，クエリを作成するために必要な情報をリクエストパラメータから取り出している．取り出される変数は action, name, password である．action は SQL のコマンド，name はユーザー名，password はユーザーのパスワードである．ソースコード 5.1 の 9 行目と 11 行目でこれらの変数を利用してクエリを作成する．ソースコード 5.1 の 12 行目で作成したク

エリがデータベースで実行される．上記のソースコード 5.1 は，リクエストパラメータを直接利用してクエリを作成しているため SQLi 脆弱性を持っている．

このソースコードに対して，クエリを実行するコードを修正する脆弱性ハンドリング関数を実装した．この脆弱性ハンドリング関数は cur インスタンスの execute メソッドが持つ引数をエスケープする escape_special_query() 関数を挿入した．escape_special_query() 関数はクエリを引数に取りクエリに drop 命令が入っている場合クエリを捨てる関数である．上記ソースコードでは 5.1 の 12 行目と 16 行目の cur.execute() メソッドの引数を escape_special_query() 関数でエスケープした．結果として，上図に示すようにソースコードは変更された．下記のソースコード 5.2 は実装された脆弱性ハンドリング関数の一部である．

```
1  class InsertQueryChecker(ast.NodeTransformer):
2    def visit_Call(self, node):
3      if isinstance(node.func, ast.Attribute):
4        if isinstance(node.func.value, ast.Name):
5          if node.func.value.id is 'cur':
6            if node.func.attr is 'execute':
7              new_args = []
8                for arg in node.args:
9                  new_arg = ast.Call(
10                   func=(ast.Name(id='
                         escape_special_query', ctx=ast.
                         Load())),
```

```
11                          args=[arg],
12                          keywords=[]
13                        )
14                    new_args.append(new_arg)
15                    new_node = ast.Call(
16                      func=node.func,
17                      args=new_args,
18                      keywords=node.keywords
19                    )
20                return ast.copy_location(new_node,
                      node)
21    return node
```

Listing 5.2: SQLi の影響を低減するための脆弱性ハンドリング関数の一部

ソースコード 5.2 の 1 行目は，クラスの宣言である．ast.NodeTransformer クラスは AST を再帰的に探索する ast モジュールが持つクラスである．このクラスの visit_属性というメソッドは特定のノードを検出した時に実行されるメソッドであり，ソースコード 5.2 の 2 行目の visit_Call() メソッドはノードの属性が ast.Call の時に実行されるメソッドである．ソースコード 3 行目から 6 行目が cur.execute() メソッドを検出するノードの条件である．その後条件に一致する AST を検出し，ソースコード 5.2 の 9 行目から 19 行目で AST を修正する．

脆弱性ハンドリング関数によって脆弱性の影響を低減できたか確認するために，まず脆弱性ハンドリング関数の実行部分をコメントアウトした

アプリケーションでローカル上の 8000 番ポート実行し，このアプリケーションに対して攻撃を行った．次に，脆弱性ハンドリング関数に解析・修正されたアプリケーションをローカル上の 8000 番ポートに立ち上げ，攻撃であるリクエストをアプリケーションに送信した．攻撃リクエストはパスが/login で，クエリの要素となるリクエストボディ部分が action=drop user;--&id=name&password=password とした．

5.1.2　不適切な認証を持つコールバック関数の修正と攻撃

不適切な認証を持つコールバック関数を VHF 上に実装した．このコールバック関数は図に示すように/login へのリクエストでは認証を行い，アプリケーション内に登録されたユーザーは”ADMIN_PAGE”が返却される．一方，/home へのリクエストは，認証なしに”ADMIN_PAGE”を返却する．このコールバック関数は適切に認証を行っていないため，全てのユーザーが”ADMIN_PAGE”を閲覧可能である．”ADMIN_PAGE”が権限を必要とするページである時，このアプリケーションは不適切な認証の脆弱性を持つ．以下のソースコード 5.3 が実装したコールバック関数である．

```
1 @app.route("^/login$", "POST")
2 def do_login(request):
3     id = request.params["ID"]
4     password = request.paramas["PASSWORD"]
5     if is_admin(id, password):
6         return "ADMIN_PAGE"
```

```
7       else:
8           return "LOGIN_PAGE"
9
10  @app.route("^/home$", "GET")
11  def home(request):
12      return "ADMIN_PAGE"
```

Listing 5.3: A vulnerable function which has an authentication leak.

このアプリケーションには2つのコールバック関数が実装された．1つ目がdo_login() 関数であり，2つ目がhome() 関数である．do_login() 関数はリクエストパスが/loginでリクエストメソッドがPOSTの時に実行される．このコールバック関数は，ユーザーのIDとパスワードによるユーザー認証を行う関数である．do_login() 関数はソースコード5.3では，2行目から8行目に該当する．3行目と4行目でユーザーのIDとパスワードをリクエストから取得する．その後，5行目で認証を行う．5行目のis_admin() 関数はユーザーのIDとパスワードを引数に取る．ユーザーIDとパスワードが一致するユーザーが存在する時Trueを返し，そうでない時はFalseを返す．ソースコード5.3のis_admin() 関数がTrueの時，do_login() 関数は"ADMIN_PAGE"を返し，Falseの時は"LOGIN_PAGE"を返す．10行目と11行目のコールバック関数はhome() 関数である．このコールバック関数はリクエストを受け取ると，"ADMIN_PAGE"を返す．

home() 関数に対して認証機能を追加する脆弱性ハンドリング関数を実装した．この脆弱性ハンドリング関数はis_admin() 関数下のノードではなく，かつis_admin() 関数下の戻り値ノードと同様のページを返すノー

ドを脆弱なノードとした．この脆弱なノードを修正するために，脆弱性ハンドリング関数は 3 つの操作を行う．まず is_admin() 関数が True である時の，属性が戻り値であるノードを検出しリストの形式にまとめる．まとめられたこれらのノードのリストをリスト A とする．次に is_admin() 関数が True である戻り値と同様のページを返すノードを検出しリストにまとめる．このリストをリスト B とする．第 3 にリスト B に含まれ，かつリスト A に含まれないノードを検出する．検出されたノードは is_admin() 関数下のノードではなく，is_admin() 関数下の戻り値ノードと同様のページを返すため脆弱である．最後に脆弱性ハンドリング関数は，検出された脆弱なノードに is_admin() 関数を追加する．

この脆弱性ハンドリング関数を評価するために，脆弱性ハンドリング関数をコメントアウトして実行したアプリケーションと脆弱性ハンドリング関数でコールバック関数を解析・修正したアプリケーションに対してそれぞれ同じ攻撃を行った．本実験の攻撃リクエストは/home へのリクエストである．

5.2　オーバーヘッドの評価

実行開始時にコールバック関数を解析・修正する時間を time() 関数を利用して計測した．コールバック関数を解析・修正するソースコードの前後に time() 関数を記述し，その差をとることでコールバック関数を修正するために必要になるオーバーヘッドを計測した．

第6章

結果

6.1　アプリケーションへの攻撃結果

脆弱性を持つ2つのアプリケーションに対して脆弱性ハンドリング関数を適用した場合と適用してない場合でそれぞれ攻撃を行った結果，脆弱性ハンドリング関数はそれぞれの脆弱性の影響を低減したことがわかった．

6.1.1　SQLi 脆弱性を持つアプリケーションへの攻撃結果

6.1.2　不適切な認証を持つアプリケーションへの攻撃結果

6.2　オーバーヘッド

hoge

第7章

おわりに

本論文ではWebアプリケーションフレームワークが持つべき機能として，コールバック関数を自動的に解析して変更する機能を提案した．提案したフレームワークを評価するために，脆弱性ハンドリング関数を持つWebアプリケーションフレームワークVHFを実装し，ローカル上で実行した脆弱なWebアプリケーションを攻撃する実験を行った．この実験から，VHFはSQLiと不適切な認証の脆弱性をアプリケーション開発者が修正することなく自動的に修正し，これらの脆弱性の影響を低減することが可能であるとわかった．また，脆弱性ハンドリング関数を実装した時の実行開始時のオーバーヘッドは小さく，実行に大きな影響を与えないことがわかった．

一方で，VHFには課題があることがわかった．それは，脆弱なノードの条件を網羅的に定義することが難しいことである．その理由は2つある．1つ目は脆弱性ハンドリング関数はコールバック関数が実装される前に実装されている必要があることである．脆弱性ハンドリング関数の

実装者はコールバック関数を実装するアプリケーション開発者のソースコードを予測して網羅的に脆弱なソースコードの条件を定義しなければならない．2 つ目は同じ意味を持ち違うノードが存在することである．あるソースコードとその糖衣なソースコードから異なる AST が生成されることがあるので，脆弱なノード全て列挙するのは困難である．したがって，今後脆弱性ハンドリング関数の効率的な脆弱性検出手法は今後の課題である．

VHF は脆弱性ハンドリング関数により，従来では対策が困難だった脆弱性の影響を低減できることがわかった．今後の改善により，VHF は安全なアプリケーションの効率的な実装へ寄与すると考えられる．