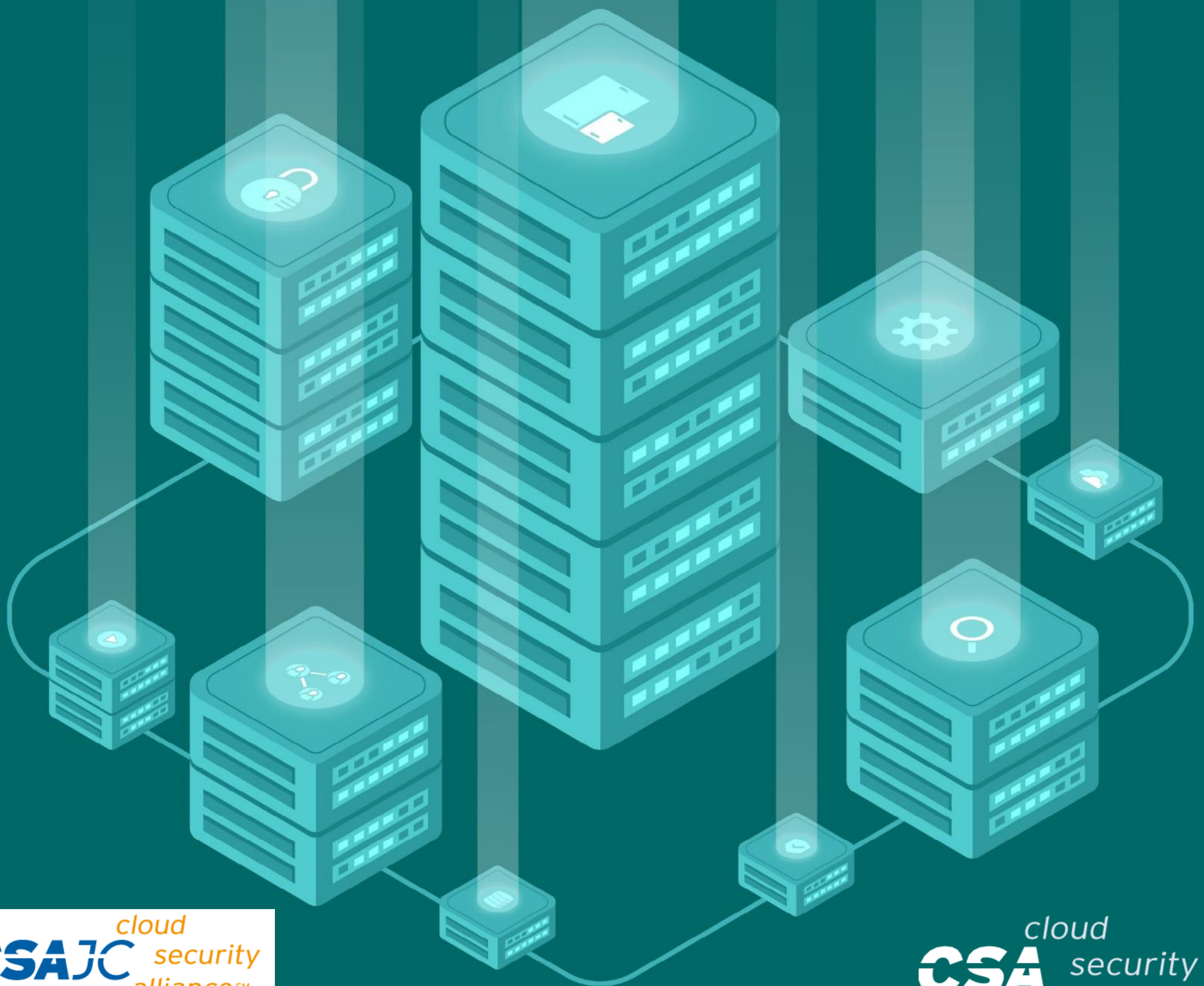
安全なサーバーレスアーキテクチャを設計するには



CSAJC cloud security alliance℠ Japan Chapter

CSA cloud security alliance®

The permanent and official location for Cloud Security Alliance Serverless Computing research is https://cloudsecurityalliance.org/research/working-groups/serverless/

© 2021 Cloud Security Alliance – All Rights Reserved. You may download, store, display on your computer, view, print, and link to the Cloud Security Alliance at https://cloudsecurityalliance.org subject to the following: (a) the draft may be used solely for your personal, informational, non- commercial use; (b) the draft may not be modified or altered in any way; (c) the draft may not be redistributed; and (d) the trademark, copyright or other notices may not be removed. You may quoteportions of the draft as permitted by the Fair Use provisions of the United States Copyright Act, provided that you attribute the portions to the Cloud Security Alliance.

Acknowledgments

Team Leaders/ Authors:

Aradhna Chetal
Vishwas Manral
Marina Bregkou
Ricardo Ferreira
David Hadas
Vani Murthy
Elisabeth Vasquez
John Wrobel

Key Contributors:

Amit Bendor Peter
Campbell Madhav
ChablaniJohn
Kinsella Namrata
KulkarniAkshay
Mahajan Eric
Matlock Shobhit
Mehta Vrettos
Moulos Raja
Rajenderan
Abhishek Vyas
Brad Woodward

CSA Global Staff:

Marina Bregkou
Claire Lehnert (Design)
Stephen Smith (Design)
AnnMarie Ulskey (Cover, Design)

Reviewers:

Anil Karmel
Alex Rebo

© Copyright 2021, Cloud Security Alliance. All rights reserved.

日本語版提供に際しての告知及び注意事項

本書「安全なサーバーレスアーキテクチャを設計するには」は、Cloud Security Alliance (CSA)が公開している「How to Design a Secure Serverless Architecture」の日本語訳です。本書は、CSAジャパンが、CSAの許可を得て翻訳し、公開するものです。原文と日本語版の内容に相違があった場合には、原文が優先されます。
翻訳に際しては、原文の意味および意図するところを、極力正確に日本語で表すことを心がけていますが、翻訳の正確性および原文への忠実性について、CSAジャパンは何らの保証をするものではありません。
この翻訳版は予告なく変更される場合があります。以下の変更履歴（日付、バージョン、変更内容）をご確認ください。

変更履歴

日付	バージョン	変更内容
2022年1月29日	日本語版1.0	初版発行

本翻訳の著作権はCSAジャパンに帰属します。引用に際しては、出典を明記してください。無断転載を禁止します。転載および商用利用に際しては、事前にCSAジャパンにご相談ください。
本翻訳の原著作物の著作権は、CSAまたは執筆者に帰属します。CSAジャパンはこれら権利者を代理しません。原著作物における著作権表示と、利用に関する許容・制限事項の日本語訳は、前ページに記したとおりです。なお、本日本語訳は参考用であり、転載等の利用に際しては、原文の記載をご確認下さい。

CSAジャパン成果物の提供に際しての制限事項

日本クラウドセキュリティアライアンス（CSAジャパン）は、本書の提供に際し、以下のことをお断りし、またお願いします。以下の内容に同意いただけない場合、本書の閲覧および利用をお断りします。

1．　責任の限定
　CSAジャパンおよび本書の執筆・作成・講義その他による提供に関わった主体は、本書に関して、以下のことに対する責任を負いません。また、以下のことに起因するいかなる直接・間接の損害に対しても、一切の対応、是正、支払、賠償の責めを負いません。
(1)　本書の内容の真正性、正確性、無誤謬性
(2)　本書の内容が第三者の権利に抵触しもしくは権利を侵害していないこと
(3)　本書の内容に基づいて行われた判断や行為がもたらす結果
(4)　本書で引用、参照、紹介された第三者の文献等の適切性、真正性、正確性、無誤謬性および他者権利の侵害の可能性

2．　二次譲渡の制限
　本書は、利用者がもっぱら自らの用のために利用するものとし、第三者へのいかなる方法による提供も、行わないものとします。他者との共有が可能な場所に本書やそのコピーを置くこと、利用者以外のものに送付・送信・提供を行うことは禁止されます。また本書を、営利・非営利を問わず、事業活動の材料または資料として、そのまま直接利用することはお断りします。
　ただし、以下の場合は本項の例外とします。
(1)　本書の一部を、著作物の利用における「引用」の形で引用すること。この場合、出典を明記してください。
(2)　本書を、企業、団体その他の組織が利用する場合は、その利用に必要な範囲内で、自組織内に限定して利用すること。
(3)　CSA ジャパンの書面による許可を得て、事業活動に使用すること。この許可は、文書単位で得るものとします。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

(4)　転載、再掲、複製の作成と配布等について、CSA ジャパンの書面による許可・承認を得た場合。この許可・承認は、原則として文書単位で得るものとします。

3.　本書の適切な管理

(1)　本書を入手した者は、それを適切に管理し、第三者による不正アクセス、不正利用から保護するために必要かつ適切な措置を講じるものとします。

(2)　本書を入手し利用する企業、団体その他の組織は、本書の管理責任者を定め、この確認事項を順守させるものとします。また、当該責任者は、本書の電子ファイルを適切に管理し、その複製の散逸を防ぎ、指定された利用条件を遵守する（組織内の利用者に順守させることを含む）ようにしなければなりません。

(3)　本書をダウンロードした者は、CSA ジャパンからの文書（電子メールを含む）による要求があった場合には、そのダウンロードしまたは複製した本書のファイルのすべてを消去し、削除し、再生や復元ができない状態にするものとします。この要求は理由によりまたは理由なく行われることがあり、この要求を受けた者は、それを拒否できないものとします。

(4)　本書を印刷した者は、CSA ジャパンからの文書（電子メールを含む）による要求があった場合には、その印刷物のすべてについて、シュレッダーその他の方法により、再利用不可能な形で処分するものとします。

4.　原典がある場合の制限事項等

本書がCloud Security Alliance, Inc.の著作物等の翻訳である場合には、原典に明記された制限事項、免責事項は、英語その他の言語で表記されている場合も含め、すべてここに記載の制限事項に優先して適用されます。

5.　その他

その他、本書の利用等について本書の他の場所に記載された条件、制限事項および免責事項は、すべてここに記載の制限事項と並行して順守されるべきものとします。本書およびこの制限事項に記載のないことで、本書の利用に関して疑義が生じた場合は、CSAジャパンと利用者は誠意をもって話し合いの上、解決を図るものとします。

その他本件に関するお問合せは、info@cloudsecurityalliance.jp までお願いします。

日本語版作成に際しての謝辞

「安全なサーバーレスアーキテクチャを設計するには」は、CSAジャパン会員の有志により行われました。作業は全て、個人の無償の貢献としての私的労力提供により行われました。なお、企業会員からの参加者の貢献には、会員企業としての貢献も与っていることを付記いたします。
以下に、翻訳に参加された方々の氏名を記します。（氏名あいうえお順・敬称略）

笹原　英司
神保　冬和子
鈴木　伸
満田　淳
諸角　昌宏
山口　弘行

© Copyright 2021, Cloud Security Alliance. All rights reserved.

目次

内容

エグゼクティブサマリ........9
1. はじめに........9
目的と範囲........9
対象読者........10
2. サーバーレスとは........11
3. なぜサーバーレスなのか........16
3.1 サーバーレスアーキテクチャの利点と効果........16
3.2 サーバーレスの責任共有モデル........17
3.3 サーバーレスが適切なのはどのような場合か........18
4. ユースケースと事例........19
5. サーバーレスのセキュリティ脅威モデル........21
5.1 サーバーレス – 全く新しいセキュリティ？........21
5.2 サーバーレスの脅威の状況........22
5.2.1 主な脅威分野........22
5.2.2 アプリケーションオーナー セットアップフェーズの脅威........23
5.2.3 アプリケーションオーナー デプロイフェーズの脅威........24
5.2.4 サービスプロバイダの行為の脅威........25
5.3 脅威モデル – アプリケーションオーナーのための25のサーバーレス脅威........25
5.4 サーバーレスの脅威の独自性........32
5.4.1 アプリケーションオーナーのセットアップフェーズの脅威（参照：5.3（A）........32
5.4.2 アプリケーションオーナーのデプロイメントフェーズの脅威（参照5.3(B)）........33
5.4.3 サービスプロバイダのデプロイメント上の脅威（参照：5.3(C)）........37
6. セキュリティのデザイン、コントロール、ベストプラクティス........38
6.1 サーバーレスの設計上の注意点........40
6.1.1 サーバーレスプラットフォームのデザインがサーバーレス・マイクロサービス・セキュリティに与える影響........41
6.2 FaaSのコントロール........46
6.3 CI-CDパイプライン、ファンクションコード、コードスキャン、ファンクションとコンテナのポリシーの実施........47
6.4 コンテナイメージベースのサーバーレスのための差分/追加コントロール........51
6.4.1 コンテナイメージベースのサーバーレスサービスにアクセスするためのAPIアクセスの管理 51

© Copyright 2021, Cloud Security Alliance. All rights reserved.

6.4.2 コンテナイメージをベースにしたサーバーレスな設定とポリシーの実施 ... 52
6.4.3 ベースイメージの管理とセキュリティ強化 ... 54
6.4.4 Kubernetesの設定とサービスメッシュポリシーの実施 ... 56
6.4.5 アクセス管理コントロール ... 58
6.4.6 Kubernetesリスクとコントロール ... 59
6.4.7 追加のセキュリティ ... 62
6.5 コンプライアンスとガバナンス ... 66
6.5.1 サーバーレスのためのアセットマネジメント ... 66
6.5.2 サーバーレスガバナンス ... 67
6.5.3 コンプライアンス ... 68
7. サーバーレスセキュリティの未来像 ... 69
7.1 サーバーレスの未来への布石 ... 70
7.1.1 サーバーレスへの動き コンテナイメージベースのサーバーレス ... 70
7.1.2 仮想化の変化 ... 70
7.1.3 FaaSの進化 ... 71
7.2 サーバーレスのセキュリティ ... 72
7.3 データプライバシーのためのサーバーレスの進歩 ... 74
8. 結論 75
Appendix 1: Acronyms ... 76
Appendix 2: Glossary ... 77

© Copyright 2021, Cloud Security Alliance. All rights reserved.

エグゼクティブサマリ

サーバーレスプラットフォームは、開発者の開発・デプロイを高速化し、コンテナクラスタや仮想マシンなどのインフラを管理せずにクラウドネイティブサービスに簡単に移行することを可能にします。企業が技術的価値をより早く市場に投入するために、サーバーレスプラットフォームは開発者の間で採用が進んでいます。

他のソリューションと同様に、サーバーレスには様々なサイバーリスクが伴います。本ドキュメントでは、サーバーレスアプリケーションのセキュリティについて、ベストプラクティスと推奨事項を中心に解説します。サーバーレスプラットフォームがさらされるアプリケーションオーナーのリスクに焦点を当てて、さまざまな脅威の概要を説明し、適切なセキュリティ対策を提案しています。

導入の観点では、サーバーレスアーキテクチャを採用している企業は、プラットフォームやコンピューティングリソースの管理や制御、ロードバランシング、モニタリング、可用性、冗長性、セキュリティなどに煩わされることなく、製品のコア機能に集中することができます。サーバーレスソリューションは本質的にスケーラブルであり、「**従量課金制**（Pay as you go）」のパラダイムに合わせて最適化されたコンピュートリソースが豊富に用意されています。

さらに、ソフトウェア開発の観点から見ると、サーバーレスアーキテクチャを採用している組織は、基盤となるオペレーティングシステム、アプリケーションサーバー、ソフトウェアランタイム環境を管理・制御する必要がない配備モデルを提供します。その結果、企業は市場投入までの時間を短縮し、全体的な運用コストを削減してサービスを展開することができます。

本ドキュメントの推奨事項やベストプラクティスは、情報セキュリティ、クラウド運用、アプリケーションコンテナ、マイクロサービスに関する豊富な知識と実務経験を持つ多様なグループによる広範なコラボレーションによって作成されました。この情報は、サーバーレス環境において何らかの責任を負う可能性のある、幅広い層を対象としています。

1. はじめに

目的と範囲

本ドキュメントの目的は、安全なサーバーレスソリューションを実装するためのベストプラクティスと推奨事項を提示することです。

サーバーレスサービスには以下の2種類のプレイヤーが含まれます：

- サービス／プラットフォームプロバイダ – サーバーレスアプリケーションが構築されるサーバーレスプラットフォームのプロバイダ
- アプリケーションオーナー – プラットフォーム上でアプリケーションを実行するサーバーレスソリューションのユーザー

© Copyright 2021, Cloud Security Alliance. All rights reserved.

サーバーレスソリューションでは、サービスプロバイダは、サービスユーザーが各実行ファイルの処理に必要なリソースを制御することなく、異なる実行ファイルのニーズに対応するために弾力的に自動割り当てされるコンピュートリソースを提供します。本ドキュメントの範囲は、ある管理機関のユーザーが、プラットフォームプロバイダとしての他の管理機関が提供するサーバーレスソリューションの上にワークロードを実装する場合に限定されています。サーバーレスの実装には、Container-as-a-Service（サーバーレスコンテナ）とも呼ばれるコンテナイメージベースのサーバーレスと、Function-as-a-Service（FaaS）とも呼ばれるファンクションベースのサーバーレスがあります。

このドキュメントでは、アプリケーションオーナーに焦点を当て、サーバーレスサービスを使用する際のアプリケーションに対する脅威を検討します。その上で、セキュリティのベストプラクティスについて具体的な提案を行い、アプリケーションオーナーが採用すべき推奨されるコントロールをリストアップしています。

さまざまなクラウドサービスのセキュリティに関する詳細は、他のドキュメントでも紹介されているため、本ドキュメントでは、サーバーレスサービスに移行することで変化する点にのみ焦点を当て、より一般的なクラウド関連のセキュリティの詳細については触れないようにします。

本ドキュメントの主な目的は、安全なクラウドコンピューティングの実行モデルとして、サーバーレスを紹介し推進することです。また、サーバーレスアーキテクチャの採用を検討しているアプリケーションオーナーの指針となることも目的としています。

対象読者

本ドキュメントの対象読者は、アプリケーション開発者、アプリケーションアーキテクト、セキュリティ専門家、最高情報セキュリティ責任者（CISO）、リスク管理専門家、システムおよびセキュリティ管理者、セキュリティプログラムマネージャー、情報システムセキュリティ責任者、その他サーバーレスコンピューティングのセキュリティに関心のある方です。

本ドキュメントは、読者がコーディングの実践、セキュリティとネットワークの専門知識、アプリケーションコンテナ、マイクロサービス、関数、アジャイルアプリケーション開発に関する知識を持っていることを前提としています。サーバーレス分野の技術は常に変化しているため、読者の皆様におかれましては、最新かつより詳細な情報を得るために、本ドキュメントに掲載されているものを含め、他のリソースを活用されることをお勧めします。

読者は、安全なソフトウェアの設計、開発、展開に関する業界標準のプラクティスに従うことが推奨されています。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

2. サーバーレスとは

a. サーバーレスの定義

サーバーレスコンピューティングとは、クラウドコンピューティングの実行モデルの一つで、クラウド事業者がアプリケーションオーナーのワークロードを処理するために必要なコンピュートリソースとインフラリソースを割り当てる責任を負うものです。アプリケーションオーナーは、自分のワークロードを処理するために、どのくらいの数のコンピュート資源をどのくらいの規模で割り当てるかを決定し、制御する必要はありません。アプリケーションオーナーは、オンデマンドでワークロードを処理するために利用可能な豊富なコンピュート資源に依存することができます。そのため、サーバーレスコンピューティングは**「従量課金制(Pay as you go)」のパラダイム**で提供され、支払いは通常、CPUの使用時間など実際の物理リソースに対して行われます。

サーバーレスを利用するアプリケーションオーナーは、実行（呼び出し、トリガ）される必要のある**「呼び出し可能なユニット」**と、呼び出し可能なユニットが実行（呼び出し、トリガ）される必要のある**「イベント」**のセットをサービスプロバイダに提供します。アプリケーションオーナーは、呼び出し可能なユニットと共に実行されるサポートコードを提供することもできます。サポートコードは、実行中の機能のコンテキスト内（同じプロセスの一部）で実行可能な「レイヤー」と呼ばれるライブラリとして提供することができます。また、同じ環境（呼び出し可能なユニットとは別のプロセス）で実行可能な「エクステンション」またはサポートスレッド/プロセスとして提供することもできます。「レイヤー」と「エクステンション」は、関数に割り当てられた同じリソースを共有し、同じコンテキストで実行されます。

「サーバーレス」という名称は、サービスを利用するアプリケーションオーナーが体験する動作にのみ適用されることに注意してください。アプリケーションオーナーからは抽象化されていますが、コードを実行する ″サーバー ″は存在しています。

b. 呼び出し可能なユニット

サーバーレスソリューションによって、呼び出し可能なユニットを提供するための様々なオプションが提供されています。一般的なサービスオプションとして、アプリケーションオーナーはサービスプロバイダがサポートするランタイム（JavaScript、Python、Javaなど）の下で機能コードを提供することができます。このようなサービスは、業界では「Function as a Service」、略してFaaSとして知られています。FaaSでは、アプリケーションオーナーから提供された機能コードは、通常、サービスプロバイダが所有・提供するコンテナイメージに埋め込まれます。FaaSの拡張として、イメージがスピンオンして機能が実行される前に、追加ライブラリのインストールやコンテナへのデータ注入によって、サービスプロバイダのイメージを拡張することもできます。

２つ目の一般的なオプションは、アプリケーションオーナーが呼び出し可能なユニットとして機能するように、彼らのコントロールのコンテナイメージを提供することを可能にします。このFaaSの拡張機能は、アプリケーションオーナーがコンテナイメージを提供し、マネージドまたはアンマネージのコンテナサービスの上で実行される非サーバーレスサービスとはよく区別されます。次の表は、この区別を強調しています。イメージベースのサーバーレスでは、サービスプロバイダは、イベントに応じて適切な数の呼び出し可能なユニットのインスタンスを実装する責任があります。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

	サーバーレス（このドキュメントで説明）		マイクロサービス 非サーバーレス（ここでは触れていません）	
名称	機能ベースのサーバーレス	コンテナイメージベースのサーバーレス	マネージドコンテナサービス	Kubernetes サービス
アプリケーションオーナーが提供する呼び出し可能なユニット	２つの依存関係を持つ関数	コンテナイメージ		
依存関係	プログラミング言語に依存しない	すべてのバイナリと依存関係がパッケージ化されているため、コードの言語に依存せずにアプリケーションを実行することができる。		
スケーリング、負荷分散、冗長化、実行ファイルのインスタンス監視の制御	サービスプロバイダ		アプリケーションオーナー	
可用性の制御、インスタンスの冗長化	サービスプロバイダは、インスタンスの可用性と冗長性をコントロールできる		アプリケーションオーナー	
基盤となるサーバーのライフサイクルとスケーリング	サービスプロバイダは、インスタンスの可用性と冗長性をコントロールできる		アプリケーションオーナー	
実行時間	通常、短時間（数秒以下）である。一般的に数分程度に制限される。		一般的には長時間、無制限。	
状態	ステートレスおよびエフェメラル – すべての状態は主に呼び出し可能なユニットの外部で維持される。		マイクロサービスの常識では、ほとんどがステートレスですが、マウントされたボリュームで状態を維持することができる。	
スケーリングコンピュート	サービスプロバイダの責任		アプリケーションオーナーの責任	
支払いモデル	従量課金制		割り当てられたリソースへの支払い	
ランタイムの責任	サービスプロバイダ	アプリケーションオーナー		
例	AWS Lambda Azure Function Google Cloud Functions IBM Cloud Functions	AWS Fargate Google Cloud Run IBM Code Engine	AWS ECS Red Hat Openshift (on AWS/Azure/...)	AWS EKS Google GKE Azure AKS IBM IKS

© Copyright 2021, Cloud Security Alliance. All rights reserved.

機能ベースのサーバーレス（別名「サーバーレス関数」）では、イメージはサービスプロバイダが所有・管理するため、イメージのセキュリティに関する責任はサービスプロバイダに委ねられます。その結果、サービスプロバイダは、イメージからアプリケーション所有者のワークロードに対するリスクを軽減するために、適切な管理を行う必要があります（5.3項の「サービスプロバイダの行為の脅威」を参照）。

コンテナイメージベースのサーバーレス（「サーバーレスコンテナ」とも呼ばれる）では、イメージはアプリケーションオーナーが所有・管理するため、イメージのセキュリティに関する責任はアプリケーションオーナーに委ねられます。その結果、アプリケーションオーナーは、イメージから自分のワークロードへのリスクを軽減するための適切なコントロールが求められることになります（5.3項の「アプリケーションオーナーのセットアップ段階での脅威」を参照）。

c. イベント

イベントとは、サーバーレス環境において、特定の機能を起動させる可能性のある条件のことで、新たな追加データ、新たなパケットの受信、または単に様々な期間の満了などが含まれます。イベントドリブンアプリケーションのアプリケーションオーナーは、サービスプロバイダにイベントのセットを提供し、呼び出し可能なユニットのインスタンスがイベントを処理するためにいつ実行される必要があるかを定義することができます。イベントのキューイング、呼び出し可能なユニットの十分なモデルの起動、サービスプロバイダが提供するSLAに基づく制限時間内に呼び出し可能なユニットによって処理される各イベントを処理する責任を負います。イベントの実際の処理時間は、課金目的のために累積されます。

イベントは、イベントのソースと各イベントのタイプに応じて異なる場合があります。イベントの例としては、タイマーイベント、Webリクエストをきっかけとしたイベント、ストレージやデータベースに保存されたデータの変更をきっかけとしたイベント、クラウドアカウント内で発生したサービス間、クラウドアカウント内のユーザーとサービス間の何らかのインタラクションをきっかけとしたイベント、クラウドアカウント内のイベントサービス、モニタリングサービス、ロギングサービスからのイベントなどがあります。

d. サーバーレスセキュリティの概要

サーバーレスセキュリティは、アプリケーションオーナーが、データの保護に加えて、アプリケーションの保護にのみ責任を負うという、新しいセキュリティのパラダイムをもたらします。サーバーの立ち上げ、OSのパッチ適用、アップデート、シャットダウンなど、サーバーやそのセキュリティの管理は、すべてサーバーレスプラットフォームプロバイダが行います。アプリケーションオーナーは、インフラに気を取られることなく、アプリケーションに集中することができます。しかし、本書の後半でアプリケーションオーナーが考慮すべき脅威について詳しく説明するように、サーバーレスには独自のセキュリティの課題があります。

さまざまなモデルにおけるプラットフォームプロバイダとアプリケーションオーナーの責任分担をより分かりやすく説明するために、次の図を作成しました。

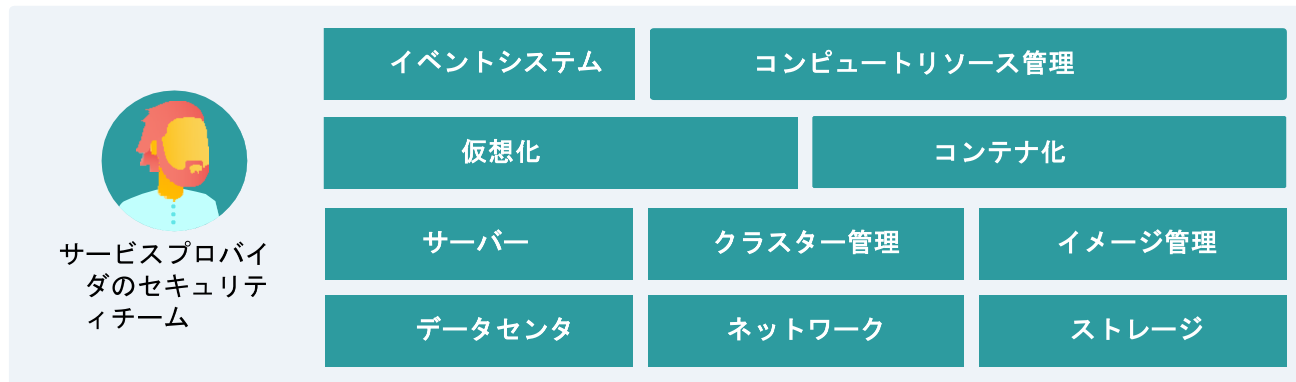
© Copyright 2021, Cloud Security Alliance. All rights reserved.

責任共有モデルの比較

サーバーレスの責任モデルの詳細図：

機能ベースのサーバーレス（FaaS）の共有責任モデル

© Copyright 2021, Cloud Security Alliance. All rights reserved.



イメーベースのサーバーレス共有責任モデル

機能ベースのサーバーレスは、運用/ランタイム環境に特化することができるため、プラットフォームプロバイダは、異なるバージョンやプログラミング言語の保守と更新の責任を負います。

e. ハイブリッドサーバーレスアーキテクチャ（プライベート＆パブリック）

サーバーレスアーキテクチャは数多く存在します。一般的なインフラストラクチャの例をいくつか紹介します（網羅的ではありません）：

- Amazon: Lambda, Fargate, AWS Batch
- Google: Cloud Functions, Knative, Cloud Run
- Azure: Azure Functions, Azure Container Instances
- Nimbella: OpenWhisk
- IBM: Knative (Code Engine), OpenWhisk (Cloud Functions)
- RedHat: Knative (OpenShift の一部)

© Copyright 2021, Cloud Security Alliance. All rights reserved.

3. なぜサーバーレスなのか

サーバーレスコンピューティングは、従来のクラウドベースやサーバー中心のインフラストラクチャに比べ、いくつかの利点があります。

サーバーレスコンピューティングでは、アプリケーションオーナーは、アプリケーションが動作するOSのメンテナンスやパッチの適用、インフラのスケールアウトなど、アプリケーションをホストするインフラを気にする必要がありません。これにより、運用のオーバーヘッドを大幅に削減することができます。[Manral, 2021]を参照してください。

さらに、アプリケーションコードは動的なプラットフォーム上でホストされ、実行されます。特定のコードは、多くの物理的なマシンのいずれかで実行される可能性があります。しかし、アプリケーションオーナーは、自分のコードが物理的にどこに存在しているかをほとんど、ほとんど知ることができません。

サーバーレスプラットフォームは通常、動的なスケーリング機能を備えています。

3.1 サーバーレスアーキテクチャの利点と効果

サーバーレスアーキテクチャの利点は、読者にわかりやすいように、以下の表で説明します。

サーバーレスは、次の利点を提供	サーバーレス （このドキュメントで説明）
デプロイの速さ	サーバーレスにより、アプリケーションオーナーはアプリケーションのインフラストラクチャを気にすることなくビジネスアプリケーションを開発することができ、ビジネスは速いペースでアプリケーションを構築・デプロイすることが可能です。したがって、実験的な試みを行ったり、新しい価値をより早くもたらすための良いツールになります。
コスト	インフラコスト。 • （通常は）イベント単位で価格設定されているため、インフラストラクチャを使用していないときには支払う必要がありません。 • バーストワークロードでの費用対効果が非常に高くなります。不要なときにサーバーを維持する必要がなく、使用するリソースを非常に細かく制御することができます。 運用コスト。 • 管理するインフラストラクチャを持たないことで、運用コストと維持に費やす時間を削減できます。[Manral, 2021]を参照。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

アプリケーションオーナーエクスペリエンス	デプロイが簡単 • サーバーレスサービスは、CLIツールやソース管理、シンプルなAPI（Application Programming Interface）から、最小限の設定で簡単に導入することができます。 モニタリングが簡単 • ほとんどのクラウドプロバイダは、サーバーレスの提供に付随して、すぐに使えるロギングとモニタリングのソリューションを提供しています。このプラットフォームはAPIドリブンであり、アプリケーションオーナーの生産性向上には欠かせません。 サーバー管理のオーバーヘッドがない • サーバーレスサービスは、パッチ適用、プロビジョニング、キャパシティ管理、運用システムの保守などのサーバー管理業務をすべて抽象化します。
スケール	本質的にスケーラブル • サーバーレスでは、インフラストラクチャを構築することなく、設定するだけで、使用量に応じて細かい粒度で自動スケーリングされます。 • ワークロードのスケールアップやスケールダウンのためのポリシーを設定する必要がありません。 • オンプレミスで作業する場合、スケーリングは利用可能なインフラストラクチャに制限されます。

3.2 サーバーレスの責任共有モデル

責任共有モデルでは、開発者は自分のコードと、クラウドにアプリケーションを提供するために使用するツールのセキュリティを確保する責任を負います。この責任共有モデルでは、各当事者は自分が所有する資産、プロセス、機能に対する完全な制御を維持します。

サービス	アプリケーションオーナー	サーバーレスプラットフォームプロバイダ
プラットフォームのパッチ適用		イメージと機能ベースのサーバーレス。
プラットフォームの構成	アプリケーションとプラットフォームに関する設定。	最小限の設定をアプリケーションオーナーに公開。
イメージのパッチ適用	コンテナイメージベースのサーバーレス。	機能ベースのサーバーレス。
セキュアコーディングの実践	イメージと機能ベースのサーバーレス。	

© Copyright 2021, Cloud Security Alliance. All rights reserved.

サプライチェーンセキュリティ	アプリケーションとコンポーネントベースのサプライチェーン。	プラットフォームサプライチェーン。
ネットワークセキュリティモニタリング		イメージと機能ベースのサーバーレス。
アプリケーションセキュリティモニタリング	イメージと機能ベースのサーバーレス。	
CI/CDパイプライン構成	イメージと機能ベースのサーバーレス。	

3.3 サーバーレスが適切なのはどのような場合か

サーバーレスモデルは、比較的大規模なアプリケーションまたはアプリケーションのセットがあり、それらをサポートするための成熟したソフトウェア開発・運用（DevOps）チーム、プロセス、製品がある場合に最も適しています。

このような場合、アプリケーションをマイクロサービスと呼ばれる小さなコンポーネントに分解することができます（「BestPracticesinImplementingaSecureMicroservicesArchitecture」を参照）。それぞれをサポートする1つまたは複数のチームによって構成され、サーバーレス環境で実行されます。これにより、特定の機能に集中することで、開発リソースをより効果的に使用することができます。また、このモデルでは、モノリシックなアプリケーションと比較して、各マイクロサービスのアジャイル開発も可能になります。なぜなら、アプリケーションの各部分の機能は、他のアプリケーション機能との完全統合やリグレッションテストをそれほど気にせずに本番環境に移行することができるからです。

比較的小規模なアプリケーションやチームの場合、サーバーレスモデルは、アプリケーションをサポートするために従来のインフラストラクチャ（IaaS（Infrastructure as a Service）やPaaS（Platform as a Service）サービスなど）よりも費用対効果が低い場合があります。一般的に、アプリケーションの規模が小さいと複雑性が低くなり、アプリケーションをマイクロサービスに分解するメリットが失われます。このような場合、マイクロサービスは他のサービスと密に結合されるため、再利用性などマイクロサービスの利点が失われる可能性があります。多くのマイクロサービスをサポートするためのリソースが不足しているため、チームはあるマイクロサービスをサポートするために、別のマイクロサービスでの作業を中断することがあります。

また、サーバーレスアーキテクチャは、ほぼすべてのケースでデプロイプロセスを簡素化します。デプロイは、コンテナイメージまたはコードのセットをアップロードするだけで、従来のアプリケーションデプロイのようにリソースプロビジョニングとネットワークアーキテクチャを気にする必要はありません。企業は、サーバーレスアーキテクチャの利用を決定する際に、ビジネスインパクト分析やコスト/ベネフィット分析を行い、ビジネスニーズに対して最も技術的に効率的で、コスト効果の高い、適切なソリューションを選択する必要があります。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

4. ユースケースと事例

サーバーレスコンピューティングの重要な役割は、クラウドプログラマーに必要なツールを提供し、インフラストラクチャを考慮する必要をなくし、プログラミング言語を直接使用できるようにすることで、クラウド指向のアーキテクチャの複雑さを軽減することです。しかし、負荷分散、リソースを効率的に利用するためのリクエストルーティング、セキュリティパッチなどのシステム更新、新しく利用可能になったインスタンスへの移行、災害時にサービスを維持するための冗長コピーの地理的分散など、多くの懸念事項に事前に対処する必要があります。

サーバーレスアーキテクチャを採用するユースケースの１つは、インフラの運用とスケーリングに幅広い技術的専門知識がなくても、巨大なスケールをもたらすことです。サーバーレスアプリケーションの構築、デプロイ、管理、スケーリングは、ハードウェア、オペレーティングシステム、サポートソフトウェアを維持することなく可能です。これにより、アプリケーションオーナーは、重要でない機能ではなく、ビジネスロジックに集中することができます。アプリケーションオーナーのコストは非常に細かく設定されているため、利用に応じて価格が比例して変化し、一貫した経済性を実現できます。しかし、総コストは運用の規模に依存する場合があります。顧客/ユーザーは、自分のニーズに基づいてプラットフォームを決定する必要があります。

サーバーレスのビジネスユースケースをいくつか紹介します。（注：これはユースケースの包括的なレポジトリになることを意図したものではなく、セキュリティ専門家にコンテキストを提供するためのいくつかの例を提供したに過ぎません）：

1. Webアプリケーション：ユーザーが既存のサービスにアクセスし、閲覧やちょっとした更新を行います。その活動が完了した後、その機能は削除されます。これは、従来のリクエストレスポンス型のワークロードです。サーバーレス機能を使用することはできますが、不適切な認証や否認防止などのセキュリティ上の脅威に対処する必要があります。
2. データ処理アクティビティはイベントドリブンであり、データ処理要求が完了した後、サービスを削除することができます。例えば、証券会社の口座を通じて、ある顧客の1日に購入したすべての口座引き落としや株のレポートを引き出すトリガーが開始されます。
3. データの完全性、機密性、セキュリティの問題は、認証や認可に加え、サーバーレス機能の一部として対処する必要があります。
4. バッチ処理のユースケースでは、一連のトリガーを設定し、データの抽出、操作、処理を行うワークフローを構築することができます。例えば、炭素排出量テストに不合格となった自動車のリストを取得し、州の要件や基準、それらを満たすための期限を記載したメールを所有者に送信します。セキュリティ面では、低権限のアクセス、データの機密性、ユーザーのプライバシーに配慮します。
5. イベントの取り込みと統合：分析アプリケーションからすべてのイベントを収集し、データベースに送り、イベントのインデックス化とファイリング、特定のトリガーによるレポート機能の開始、ダッシュボードやウェブインターフェイスへの公開を行います。このような場合、セキュリティの観点からアクセス管理および否認防止に対処し、検知に必要なメタデータを含むログが生成されるようにする必要があります。
6. サーバーレスは、業界ではすでにセキュリティ検知や自動応答に利用されており、設定ミス警告とその後のアクションに大きく貢献しています。
7. サーバーレスは、画像認識や処理にも利用できます。例としては、Google Vision や Amazon Rekognition のサービスがあり、その後、識別に基づいてそれらの画像をインデックス化します。
8. あるいは、顧客がクレジットカード情報をアップロードし、属性を抽出してトランザクションを処理するようなアプリでもサーバーレスが利用できます。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

9. このようなユースケースでは、データセキュリティ、プライバシー、アイデンティティ、アクセス管理に関する懸念に対処する必要があります。
10. トリガーを生成して、SaaSプロバイダにイベントをパイプし、処理させるようなユースケースもあります。
11. CI-CDパイプラインには、ソフトウェア開発を迅速に反復する機能が必要です。サーバーレスはこれらのプロセスの多くを自動化することができます。コードチェックはビルドと自動化された再デプロイのトリガーとなり、変更要求は自動テストの実行のトリガーとなり、人間のレビュー前にコードが十分にテストされていることを確認することができます。自動化が必要なところはどこでも、サーバーレスアプリケーションで簡素化または高速化できる可能性があり、ワークフローから手動タスクを簡単に排除することができます。
12. IoTセンサーがメッセージを入力する産業環境では、設定可能なトリガーに基づいてメッセージを処理し、サーバーレス関数を使用してメッセージに応答し、応答をスケールする必要があります。しかし、安全でない機能の影響は、場合によっては深刻なものとなります。そのため、認証、データ保護、検出は、失敗に加えて、これらのシナリオで対処する必要がある重要なコントロールです。
13. ビジネスロジックに基づいて特定のステップを実行する必要があるアプリケーションがある場合、一連のステップを実行するマイクロサービスのワークロードのオーケストレーションは、サーバーレス機能を使用して実装することができます。サーバーレスの利用には、オーケストレーションの観点、監査可能性の観点からトリガーとなるイベントによって渡されるデータ/メタデータ、障害/可用性、認証/認可に加えて否認防止の問題など、いくつかのセキュリティ上の懸念事項があります。
14. ホリデーシーズンにおけるカスタマーサービスのユースケースでは、顧客からの問い合わせに対応します。チャットボットはサーバーレスで、ピーク時の需要に合わせて自動的にスケーリングし、チャットの終了後に機能を削除する機能を提供します。
15. ユーザー認証とデータ保護/プライバシーは、依然としてセキュリティの観点から対処が必要な懸念事項です。
16. 業界におけるサーバーレスのその他の一般的なユースケースは、スケジュールされた時間でのバックアップなどのインフラ自動化タスクです。

企業におけるセキュリティチームの観点からの重要な問題は、プラットフォーム内のすべてのサーバーレス機能がどこで使われているのかが分からないことです。次の章では、エンタープライズにおけるサーバーレスの可視化と資産追跡に役立つコントロールの構築について、詳細な推奨事項を説明します。

上述したように、セキュリティチームは、セキュリティタスクの自動化において、サーバーレス機能をトリガーにして使用することができます。検出と応答のユースケースはすでに業界で普及していますが、セキュリティチームがサーバーレスを利用できる他のタスクは、自動スキャンと、CI-CDパイプラインに脆弱性がある場合のビルドの破壊を強制することです。

サーバーレス機能は、インフラストラクチャのオンデマンドスキャンをトリガーし、その結果を報告したり、その自動実行を行ったりすることができます。

サーバーレス関数は、他のセキュリティ制御の実施にも使用できます。たとえば、ユーザーがデータ要素をアップロードするたびに、関数がそのデータにタグ付けまたはラベル付けがされているかどうかをチェックします。タグ付けされていない場合、アップロードされたファイルにタグ付けされていないことを示すアラートまたは電子メールがユーザーに送信されます。そして、適切な DLP やその他のコンプライアンスポリシーを実施できるように、ラベル付けされるまで隔離されます。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

サーバーレス関数は、例えば、サービスへのアクセスを検証する際に、ある関数が認証の保証を実行できるかどうかをチェックするような、セキュリティポリシーの実施に使用することができます。ステップアップ認証が必要な場合もあるので、認証の保証レベルに失敗した場合、別の機能がステップアップした認証を要求するトリガーとなります。また、同様に、セキュリティポリシーを実施する際に使用される別の関数が存在する可能性もあります。

同様に、セキュリティにおけるサーバーレスの利用も検討されており、サーバーレスを利用してセキュリティタスクを簡素化、自動化できる可能性は大いにあります。同時に、セキュリティチームと開発者は、次の章で詳しく説明するサーバーレス技術に不可欠な脅威の状況とセキュリティ制御を理解することが適切です。

5. サーバーレスのセキュリティ脅威モデル

サーバーレス技術を使用する際、アプリケーション開発者が考慮しなければならない多くの脅威があります。このセクションでは、サーバーレスサービスを使用することで、アプリケーションサービスの脅威の状況がどのように、そしてなぜ変化するのかについて説明します。固有の脅威を含む詳細な脅威モデルとそれらがサーバーレス技術にどのように現れるか、そしてサーバーレスのための特別なセキュリティ上の注意とツールについて説明します。サーバーレスの緩和策、アーキテクチャデザイン、セキュリティコントロールの緩和については、第6章に続きます。

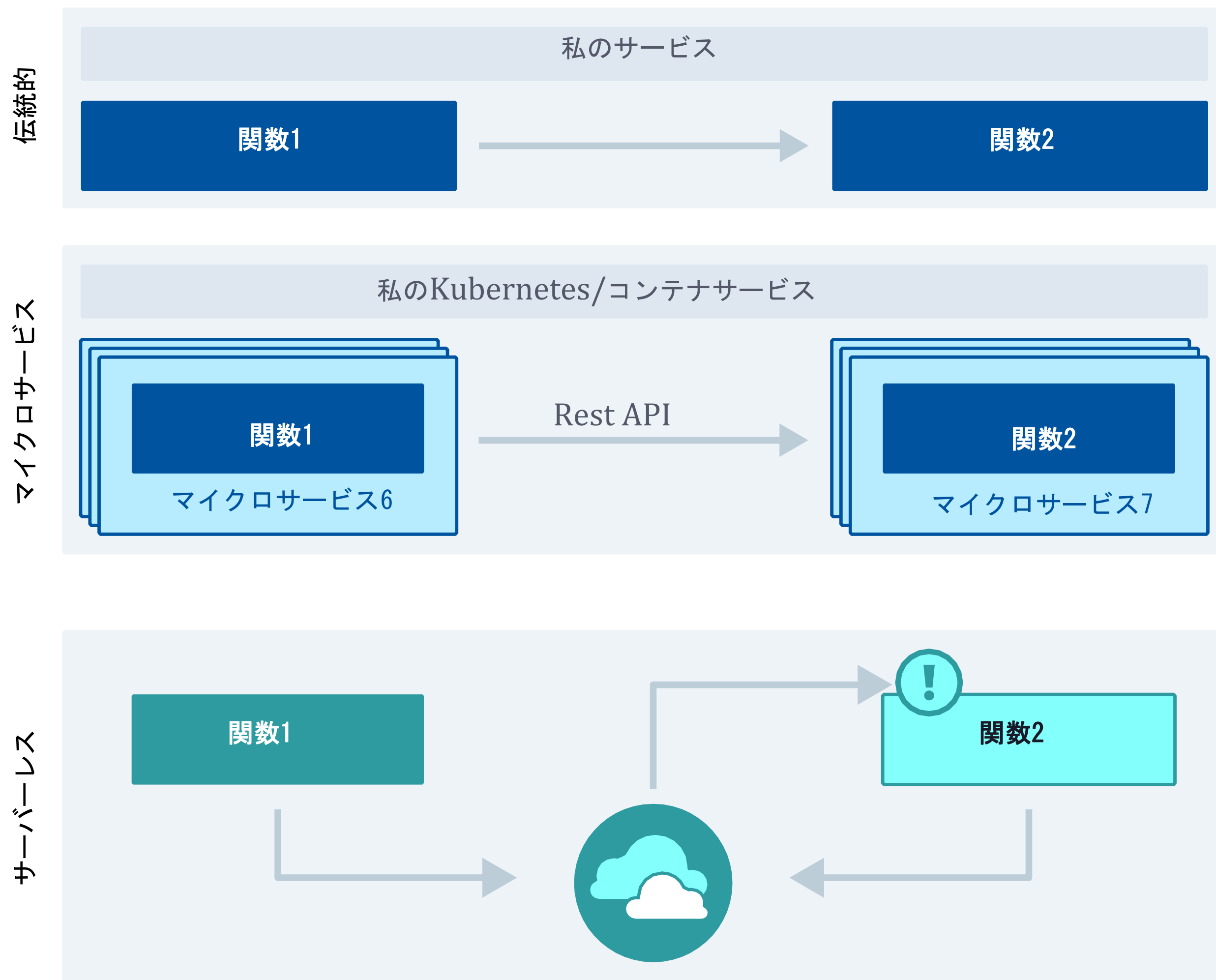
5.1 サーバーレス – 全く新しいセキュリティ？

クラウドサービス事業者によるサーバーレスサービスの導入は、アプリケーションやサービスの開発者や所有者に新たなセキュリティ上の課題をもたらします。サーバーレスは、イベント駆動型のアーキテクチャとして、ワークロードを、シームレスに分離された複数の実行環境に分割します。それぞれが特定のタスクを実行し、個々のイベントを処理し、時間と空間も別々に実行され、独自の依存関係、コード、イメージ、権限の要件、構成、寿命などがあります。これは、従来のセキュリティ脅威モデルや、サーバーレスではないマイクロサービス環境におけるクラウド脅威モデルとは大きく異なるものです。サーバーレスを利用するアプリケーション所有者は、進化した脅威の状況を再評価し、適切なセキュリティコントロールを再考する必要があります。

サーバーレスには、公の場所からデータを取得する、お客様の環境にデータを入力する、他の場所にある機能を呼び出すなど、信頼関係の境界を越えるフローが含まれる場合があります。既存のネットワークの境界が薄れていきます。この断片化により、攻撃を検知するために収集、処理、分析するセキュリティ生データの量を大幅に増やす必要が出てくるかもしれません。

サーバーレスは、ワークロードをクラウドに深く浸透させるためのもう一つのステップであり、これまでワークロードの開発者が所有しコントロールしていた機能を、クラウドサービス事業者（CSP）に移行させます。例えば、従来のコードでは関数呼び出しとして実装され、マイクロサービスではRest APIとなる可能性があったものが、今ではイベントを送信し、キューイングし、CSPによって処理されるようになります。CSPは、実際のデータ処理が、ある関数で完了するまで、つまり要求者がこのデータの受け渡しを要求した後しばらくの間、これを見守ることになります。このプロセスでは、サーバーレス・マイクロサービス／アプリケーションのセキュリティを確保するために、サービス利用者が誰に何をしてもらうか、何をすべきかが再構成されます。

© Copyright 2021, Cloud Security Alliance. All rights reserved.



サーバーレス – 全く新しいセキュリティ

5.2 サーバーレスの脅威の状況

5. 2. 1主な脅威分野

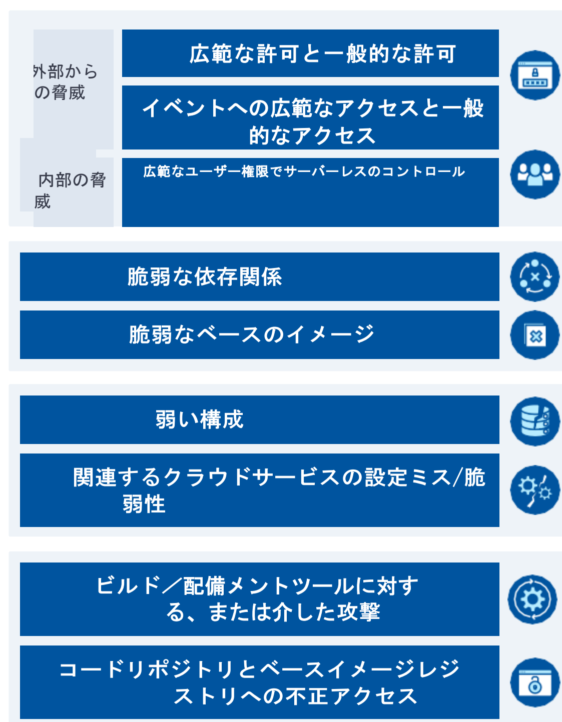
サーバーレスを使用する際のアプリケーションオーナーのワークロードへの脅威は、以下のように分けられます。

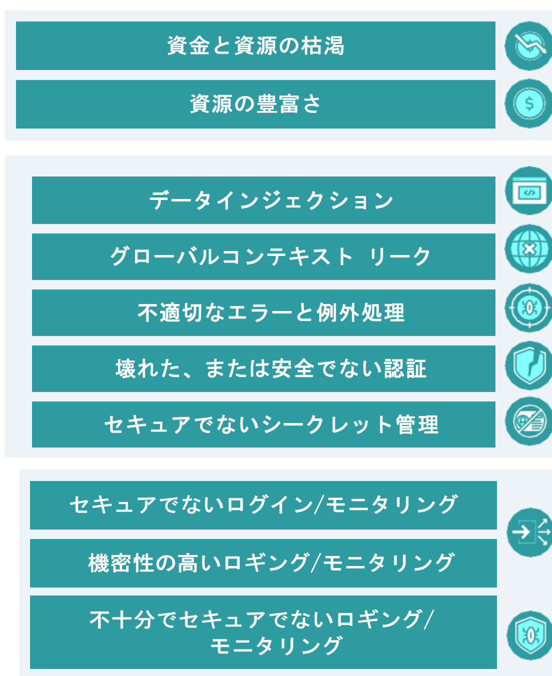
1. アプリケーションオーナー セットアップフェーズ の脅威
2. アプリケーションオーナー デプロイフェーズ の脅威
3. サービス事業者の行為による脅威

下の図では、サーバーレスの下でアプリケーションオーナーが直面する主な脅威分野をまとめています。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

セットアップフェーズ　　　　　デプロイフェーズ





脅威をもたらす行為

脅威の状況

5.2.2 アプリケーションオーナー セットアップフェーズの脅威

アプリケーションオーナーがワークロード資産を準備し、必要なコード、イメージ、CI/CD作業、クラウドリソースのプロビジョニングと設定などを含めてデプロイすることに関連する脅威のセットを次のように呼びます。- **アプリケーションオーナーのセットアップ段階の脅威**。これらには、サーバーレス特有の脅威と、マイクロサービスのようなシステムに存在する脅威の両方が含まれます。サーバーレスを使用すると、時間の経過とともにシステム内で実行または起動する異なるコード断片の数が予想以上に増えます。

アプリケーションオーナーのセットアップフェーズの脅威では、サーバーレスならではのアクセス許可や設定ミスによる脅威があります：

- 広範な許可と一般的な許可
- イベントへの広範なアクセスと一般的なアクセス
- 広範なユーザー権限でサーバーレスをコントロール
- 脆弱な構成

アプリケーションオーナーのセットアップフェーズの脅威は、サーバーレスへの移行に伴い、以下のように悪化します。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

CI/CDパイプラインまたは依存関係にあるデプロイパイプライン関連の脅威：

- リポジトリとベースイメージレジストリへの不正アクセス
- ビルド/デプロイメントツールに対する、または介した攻撃
- 脆弱な依存関係
- 脆弱なベースイメージ

設定ミスによるサービスセットアップ関連の脅威：

- 関連するクラウドサービスの設定ミスや脆弱性

5.2.3アプリケーションオーナー デプロイフェーズの脅威

サーバーレスに関連するすべてのクラウドおよびオフクラウドの資産を含む、アプリケーションオーナーによるワークロード資産のデプロイに関連する一連の脅威を、「**アプリケーションオーナー デプロイメントフェーズの脅威**」と名付けます。ここでも、サーバーレスに特有の脅威と、サーバーレスの使用に伴って悪化する脅威に分けています。

アプリケーションオーナー のデプロイフェーズでのサーバーレス特有の脅威は以下の通りです：

呼び出し可能なユニットの設計と実装によってもたらされるランタイム関連の脅威：

- データインジェクション
- グローバルコンテキストリーク
- 不適切なエラーと例外処理
- 壊れた、または安全でない認証

サーバーレスの特徴である従量課金に関する脅威：

- 資金と資源の枯渇（資源の制限が設定されている場合）
- 資源の豊富さと意図しない出費

アプリケーションオーナーのデプロイフェーズでサーバーレスを利用することで悪化する脅威には以下のようなものがあります：

- セキュアでないシークレット管理
- セキュアでないロギング/モニタリング
- ログやメタデータに含まれる機微なデータ
- 不十分でセキュアでないロギング/モニタリング

© Copyright 2021, Cloud Security Alliance. All rights reserved.

5.2.4サービスプロバイダの行為の脅威

アプリケーション所有者が利用するサービスプロバイダのサービスに関連する一連の脅威に名前を付けます。これには、サービスプロバイダが使用するスタック全体とサーバーレスサービスまたは関連サービスを形成するために使用される依存関係、個人的な資産、その他の資産が含まれます。**サービスプロバイダの行為の脅威**。この脅威は、サーバーレスに特有のものですが、他のサービスにも類似点があります：

- 脆弱性/悪意のあるサービスのベースイメージ
- 脆弱なサービスランタイム
- 呼び出し可能なユニットの呼び出し中に起こる漏洩
- 異なる呼び出し可能なユニット間の漏洩
- サーバーレスサービスの正しさ
- API/ポータル/コンソールの脆弱性

5.3 脅威モデル – アプリケーションオーナーのための25のサーバーレス脅威

サーバーレスの様々な脅威を以下の表に示します：

番号	脅威の概要	脅威の説明	緩和策（セキュリティコントロール）
アプリケーションオーナー セットアップフェーズの脅威(A)			
サーバーレスに特有のもの			
1	**広範かつ誰でもアクセス可能** 呼び出し可能なユニットの最小特権原則を維持していないこと。	アプリケーションオーナーは、各サーバーレスの呼び出し可能なユニットが実行中に持つ一連の権限を定義することができます。過剰な許可は、攻撃の一部として利用することができます。	セクション6を参照。 6.3.5, 6.3.7.1, 6.3.7.2.など。
2	**イベントへの広範かつ誰でもアクセス可能** 最小特権原則を維持していない呼び出し可能なユニットのトリガーとなるイベントが開始されること。	アプリケーションオーナーは、呼び出し可能なユニットのトリガーとなるイベントを誰が起こすかを定義できます。広範なアクセスにより、攻撃の実行が大幅に簡素化されます。特にイベント駆動型のサーバーレスアーキテクチャでは、このことが攻撃対象領域に影響を与えます。	セクション6を参照。 6.3.5, 6.3.7.1, 6.3.7.2.など。
3	**広範なユーザー権限でサーバーレスをコントロール** DevOpsチームにおいて最小特権原則を維持していない。	アプリケーションオーナーは、サーバーレスサービスやイメージストアなどをセットアップするためのアクセス権を持つ人を定義することができます。 広範なアクセスにより、潜在的な経路が増え、攻撃者又はインサイダーによって攻撃されるリスクも高まります。	セクション6を参照。 6.3.5, 6.3.7.1, 6.3.7.2.など。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

4	**脆弱な構成** サーバーレスの構成管理ミスや構成ドリフトにより、プラットフォームや常駐アプリケーションが脆弱になる可能性があります。	サーバーレスアプリケーションをホストするためのサービスの多くは、セキュアでない構成になっています。特定の設定パラメータは、アプリケーションの全体的なセキュリティポスチャに重大な影響を与えるため、注意を払う必要があります -- 例えば、誰が機能を実行するための役割を引き受けることができるか、その想定される役割に基づいて、何を実現できるのかなどです。	セクション6を参照してください。 6.3.7.2. 6.3.2 6.3.3 6.3.4など。
	サーバーレスを悪化させるもの		
5	**関連するクラウドサービスの設定ミスや脆弱性** ワークロードを構築するためにサーバーレスサービスと協調して動作する追加のクラウドサービス自体に設定ミスや脆弱性があるかもしれません。	多くの場合、呼び出し可能なユニットのセキュリティは、使用されている関連クラウドサービスのセキュリティに依存しています。たとえば、呼び出し可能なユニットは、シークレットサービスのセキュリティやアイデンティティ／アクセス管理（IAM）システムのセキュリティなどに依存している場合があります。また、呼び出し可能なユニットは、サプライチェーンの一部として、サードパーティが所有するサービスに依存することがあります。したがって、他のサービスへの依存や、サーバーレスアプリケーションの一部であるリソースとして使用されているこれらのサービスにおける設定の誤りは、サーバーレス機能の完全性に影響を与える可能性があります。	セクション6、6.3.7.1参照。 6.3.7.2, 6.3.3, など。
6	**リポジトリとベースイメージレジストリへの不正アクセス** ライブラリの依存関係やベースイメージの保存に使用されるリポジトリやレジストリの脆弱性。	攻撃者は、共有されている（公開または非公開の）コードリポジトリや画像イメージレジストリの脆弱性を見つけて、悪意のあるコードを組み込もうとする可能性があります。独立した呼び出し可能なユニットの数が大幅に増える可能性があるため、この脅威はサーバーレスでは増大します。	セクション6を参照。 6.3.3など。
7	**ビルド/デプロイツールに対する、またはそれを介した攻撃** 呼び出し可能なユニットとイベントのビルドやデプロイに使用されるCI/CDオートメーションツールの脆弱性や設定ミス。	CI/CDプラクティスの一環として、呼び出し可能なユニットを構築し、それを（トリガーとなるイベントを含めて）デプロイするために、自動化されたツールがしばしば使用されます。このような自動化では、呼び出し可能なユニットを保存したり、サーバーレスのクラウドサービスを設定したりするための昇格した権限をツールに与える必要があります。攻撃者は、これらの昇格した権限を使用して、ターゲットアプリケーションに悪意のあるコードを組み込んだり、サーバーレスアプリケーションの更新に関するDoS攻撃を引き起こす手段として使用することができます。	セクション6を参照。 6.3.7.1など。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

8	**脆弱な依存関係** 呼び出し可能なユニットのサードパーティ製ライブラリの脆弱性や悪意のあるコードは、サプライチェーン攻撃につながる可能性があります。	アプリケーションオーナーの呼び出し可能なユニットは、複数のサードパーティライブラリに依存していることがよくあります。このようなライブラリには、既存の脆弱性や新たに発見された脆弱性が含まれている可能性があります。さらに、悪意のある貢献者がそのようなライブラリにマルウェアを埋め込む可能性もあります -- これは、サーバーレス・マイクロサービスを含むすべてのアプリケーションとサービスに当てはまります。	セクション6を参照。 6.1, 6.2, 6.3.7.1, 6.3.7.2など。
9	**ベースイメージの脆弱性** イメージベースのサーバーレスサービスのイメージを形成するために使用されるベースイメージの脆弱性があります。	アプリケーションオーナーがイメージベースのサーバーレスでイメージを構築するために使用するベースイメージは、プリインストールされた依存関係にある既存または新規に発見された多くの種類の脆弱性の影響を受けやすく、プリインストールされたマルウェアを含む可能性もあります。	セクション6を参照。 6.3.3など。

アプリケーションオーナー デプロイメントフェーズの脅威(B)			
サーバーレスに特有のもの			
1	**データインジェクション** サーバーレス呼び出し可能なユニットは、起動時に様々なイベントからの入力を受け取りますが、そのようなイベントはそれぞれデータインジェクションの潜在的な脅威となります。	インジェクションの欠陥は、信頼されていない入力がインタプリタに直接渡されたり、適切に審査され検証される前に実行されたりすることで発生します。このような欠陥は、しばしば攻撃の一部となります。ほとんどのサーバーレスアーキテクチャは、データインジェクション攻撃の潜在的なベクトルとして、多数のイベントソースを提供しています。イベントデータインジェクションは、ビジネス機能を実行するための機能のオーケストレーションを壊し、サービス拒否を引き起こす可能性もあります。	セクション6を参照。 6.3.7.2 など。
2	**グローバルコンテキストのリーク** サーバーレスのグローバルコンテキストでは、例えば、トークンを横断的に管理することができます。呼び出し可能なユニットの呼び出し（たとえば、呼び出しごとにID管理機能に対して再認証する必要性を省くことができる）。グローバルコンテキストは、リクエスト間で機密データを漏洩する可能性があります。	呼び出し可能なユニットの異なる呼び出しは、ワークロードの他のユーザーが所有するデータを提供するためにしばしば使用されるので、呼び出し可能なユニットの追加要求間のデータ漏洩は脅威です。機密性の高いデータがコンテナ内に残されている可能性があり、その後の関数の呼び出しの際に公開される可能性があります。悪意のあるデータが意図的に残され、今後の関数の呼び出しの際に攻撃される可能性があります。	セクション6を参照。 6.3.7.2など。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

3	**不適切なエラーと例外処理** サーバーレスベースのアプリケーションに対するクラウドネイティブのデバッグオプションは、標準的なアプリケーションのデバッグ機能と比較すると、制限されているか、複雑になっています。	不適切なエラー処理は、脆弱性を生み出し、バッファオーバーフロー攻撃やDoS攻撃などの悪意ある行為を許す可能性があります。冗長なエラーメッセージにより、攻撃者に意図しない情報が開示される可能性があります -- これはメタデータやリソースの公開という点では、すべてのアプリケーションやサーバーレスにも当てはまります。	セクション6を参照。 6.3.7.2など。
4	**壊れた、またはセキュアでない認証** イベントのソースのアイデンティティおよび当該イベントを開始するユーザー/プロセスのアイデンティティの不適切な認証。	多くの場合、呼び出し可能なユニットは、送信されるイベントの背後にあるエンティティのアイデンティティを確認する必要があります。攻撃者は、使用されている認証メカニズムの脆弱性を利用しようとします。	セクション6を参照。 6.3.7.2, 6.3.5など。
5	**資金とリソースの枯渇** 攻撃者が計画外の多額の出費を引き起こす仕組みとしてのサーバーレス	攻撃者は、サーバーレスが「従量課金」サービスであることを利用し、アプリケーションオーナーの呼び出し可能なユニットを呼び出す偽のイベントを多数作成したり、処理時間が長くなるイベントを開始したりすることで、アプリケーションオーナーに予定外の多額の費用を負担させる可能性があります（例えば、コードや依存関係にある他の弱点を露呈させるなど）。	セクション6を参照。 6.3.6など。
6	**リソースの豊富さ** 無限のコンピュートリソースを利用するための仕組みとしてのサーバーレス	攻撃者は、サーバーレスが無制限のリソースプールとして提供されていることを利用するかもしれません。このような脆弱性を考慮すると、攻撃者は以下のような動機付けを受けることにもなります。呼び出し可能なユニットで利用可能な豊富なコンピュータを利用して、例えば、クリプトマイニングや第三者への攻撃を開始するなど、自分の利益のために利用すること。	セクション6を参照。 6.3.7.1 6.3.7.2, 6.3.3,など。
サーバーレスを悪化させるもの			
7	**不十分でセキュアでないロギング/モニタリング** セキュリティインシデントに対する状況認識が不十分で、セキュリティ侵害の調査ができないこと。	ロギングが不十分だと、組織が攻撃や侵害に迅速に対応することができなくなり、フォレンジック分析を行うことが困難または不可能になります。	セクション6を参照。 6.3.7.1 6.3.7.2, 6.3.3,など。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

8	**セキュアでないシークレット管理** 呼び出し可能なユニットで使用されているシークレットが漏洩し、システムの一部に意図せずにアクセスされたり、特権の昇格が可能になったりする可能性があります。	呼び出し可能なユニットは、呼び出される時に特定のクラウドや外部リソースにアクセスする必要があることがよくあります。そのために、呼び出し可能なユニットはシークレットを取得する必要がある場合があります。攻撃者は、シークレットが安全に保存されていない場合や、標準的な資格情報のセットが使用されている場合などの状況を利用することができます。サーバーレス機能における他のアプリケーションと同様に、資格情報やシークレットの漏洩は、なりすましによるIDやデータの漏洩につながる可能性があります。	セクション6を参照。6.3.7.1 6.3.7.2など。（参照：#3.過剰なデータ漏洩）
9	**セキュアでないロギング/モニタリング** ロギングのデータを攻撃者に公開したり、攻撃者がログを削除できるようにすること。	セキュアでないロギングにより、攻撃者が攻撃のトラブルシューティングを可能にしたり、行動の痕跡を削除して発見やフォレンジックを妨げたりする可能性があります。同時に、関数所有者がセキュリティ問題を検知せず、対応しないこともあり、サーバーレスアプリケーションの全体的な完全性に影響を与えます。	セクション6を参照。6.3.7.1 6.3.7.2, 6.3.3, など。
10	**機密性の高いロギング/モニタリング** セキュリティやプライバシーに関わる機密情報をロギングします。	呼び出し可能なユニットやイベントは、ロギングシステムやモニタリングシステムを介して、機密情報、PII、ユーザーデータなどの機密性の高いデータを公開する可能性があります。	セクション6を参照。6.3.7.1 6.3.7.2, 6.3.3, など。

サービスプロバイダの行為の脅威(C)			
サーバーレスに特有のもの			
1	**脆弱性/悪意のあるサービスのベースイメージ** ファンクションベースのサーバーレスでは、サービスプロバイダが選択したベースイメージに脆弱性/マルウェアが含まれている可能性があります。	サービスプロバイダが使用するイメージには、複数のサードパーティの依存関係があります。このようなイメージは、既存または新たに発見された脆弱性の影響を受けやすく、プレインストールされたマルウェアが含まれている可能性があります。サービスプロバイダがプラットフォームの基礎となるスタックを管理していることを考えると、これはサーバーレスアプリケーションのセキュリティポスチャに影響を与える可能性があります。	セクション6を参照。6.3.3

© Copyright 2021, Cloud Security Alliance. All rights reserved.

2	**脆弱なサービスランタイム** ファンクションベースのサーバーレスでは、ランタイムはサービスプロバイダのコードによって構成され、しばしば拡張されるため、脆弱なランタイムになる可能性があります。	ファンクションベースのサーバーレスでは、サービスを形成したり、サービスプロバイダを監視したりするために、ベースイメージが追加コードで拡張されることが多くなります。デプロイ時には、実行中のコンテナの構成はサービスプロバイダによって設定されます。これにより、ポートの開放や実行環境に統合されたその他の管理機能など、潜在的な脆弱性が発生する可能性があります。したがって、サーバーレスアプリケーションのセキュリティコントロールは、完全なコンテキストで評価することが適切です。	セクション6を参照。6.3.6 (Kubernetes リスク＆コントロール：ディープダイブ) 6.3.7.1など.
3	**呼び出し可能なユニットの呼び出し中に起こる漏洩** 与えられた呼び出し可能なユニットの呼び出し間の分離を介したサーバーレスサービスの脆弱性は、定義されたサーバーレスサービス契約外で破られます。	呼び出し可能なユニットの異なる呼び出しは、複数のワークロードユーザーが所有するデータを提供することが多いため、呼び出し中のデータの漏洩は大きな脅威となります。 例えば、異なるエンドユーザーやセッションコンテキストに対応するために再利用された呼び出し可能なユニットは、残された以前のユーザーの機密性の高いデータを漏洩する可能性があります。また、意図的に残された悪意のあるデータ/状態が、後続のユーザーに害を及ぼす可能性もあります。	セクション6を参照。 6.3.7.2など。
4	**異なる呼び出し可能なユニット間の漏洩** 同じ実行環境のインスタンスの上で順番に実行される分離された呼び出し可能なユニットを介したサーバーレスサービスにおける脆弱性で破られます。 例：関数1が終了し、FaaSの下で関数2が同じ実行環境で、適切なクリーンアップなしでロードされた場合。	他の呼び出し可能なユニットは、異なるクラウドおよびデータへのアクセス権限を持ち、異なるアイデンティティおよびシークレットを保持し、様々な悪用可能な脆弱性などを持つため、異なる呼び出し可能なユニット間の漏洩は大きな脅威となります。 例えば、サーバーレスの実行環境を再利用して、ある呼び出し可能なユニットから別の呼び出し可能なユニットへとサービスを提供した場合、機密データが残されたり、悪意のあるデータ/状態が意図的に残されて、後続のユーザーに危害を加えたり、後続の特権を利用したり、アイデンティティを侵害したりする可能性があります。 また、後続の呼び出し可能なユニットのシークレットや脆弱性を利用することもできます。	セクション6を参照。 6.3.7.1など

© Copyright 2021, Cloud Security Alliance. All rights reserved.

5	**サーバーレスサービスの正しさ** サーバーレスでは、ワークロードは、アプリケーションオーナーのコードの断片を組み合わせたものです。したがって、サーバーレスのコアサービスの正しさは、ワークロードの正しさに直接影響する。	サーバーレスではないマイクロサービス型のクラウドサービスとは異なり、ワークロードのセキュリティは、サービスプロバイダのイベントシステム、イメージ管理、呼び出し可能なユニットの実行インスタンスへのアクセスコントロールが正しいかどうかに依存します。これらのシステムが特定の条件下で障害を起こすと、攻撃者の侵入を許すことになります。例えば、誤ったイベントを送信したり、不適切な機能やコンポーネントを開始したり、誤った権限を与えたりすることなどです。そのため、認証や認可に加えて、機能のオーケストレーションや実行前の主要イベントの検証が不可欠です。	セクション6を参照。 6.3.2, 6.3.4, 6.3.7.1など
6	**API/ポータル/コンソールの脆弱性** Web API、throw CLI、またはWebインタフェースを使用して、ユーザーがサーバーレスプラットフォームをリモートで設定および管理できるようにする脆弱なAPIです。	管理ポータルやコンソールの場合、複数のレベルでプラットフォームへの不正アクセスが発生し、その結果、設定の変更（および弱体化）や環境の偵察活動を行うことができてしまいます。他のアプリケーションがAPIを通じてサーバーレスアプリケーションを呼び出すことがあります。サーバーレス機能の一部として、追加のリソースやサービスを呼び出すことがあります。したがって、セキュアなAPI、その入力と出力、およびコンテキストは、サーバーレス機能全体のセキュリティに不可欠です。	セクション6を参照。 6.3.6, 6.3.1, 6.3.7.1など。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

5.4 サーバーレスの脅威の独自性

このセクションでは、上記の脅威をサーバーレスアプリケーションの文脈としてさらに説明します）。

このセクションでは、他のIT環境と比較したサーバーレスの側面と、前のセクションで特定された脅威にサーバーレスがどのように影響するかについて説明します。これは、サーバーレス環境に関連する特定の脅威の重要性を詳しく説明し、強調するためです。サーバーレス環境における脅威を他のIT環境と比較することによって、サーバーレスセキュリティの独自性が明らかになってきます。

このセクションで述べられているすべての脅威は、太字で表示され、前のセクションで説明されています。

5.4.1アプリケーションオーナーのセットアップフェーズの脅威（参照：5.3（A）

このサブセクションでは、セットアップフェーズの脅威に関連した、サーバーレスとその他の環境の違いを詳しく説明し、セキュリティ担当者や実務者が特に注意を払う必要がある側面を特定します。

フラグメンテーションの側面

最小特権の原則を維持することは、設定可能なエンティティやパラメータが膨大な環境である場合、特に困難です。サーバーレスサービスを使用する場合、ワークロードは小さな呼び出し可能なユニットに分割され、それぞれにセキュリティを管理する一連のパラメータが設定されます。これらのパラメータには、ユーザーの認証情報やアクセス権、呼び出し可能なユニットの変更や設定を許可されたシステム、呼び出し可能なユニットが実行時に使用する認証情報とアクセス権、呼び出し可能なユニットに関連するロギング、モニタリング、使用状況、その他の通知の制御などが含まれます。これにより、**「幅広く包括的な許可」**、**「幅広く包括的なイベントへのアクセス」**、**「サーバーレスコントロールに渡る幅広いユーザー特権」**による脅威からの保護を求めるアプリケーション所有者にとって、新たな課題が生じます。

さらに、複数のフラグメントの安全な構成を作成し、その後維持することも課題となります。フラグメントの数が多いと、上記で特定された脅威である**「脆弱な構成」**や **「関連するクラウドサービスの設定ミスや脆弱性」**への対応が複雑になります。

アプリケーションの所有者が、すべてのコンポーネントに対して安全な構成を確保し、本番環境への初期導入時には十分に制限された特権セットを使用したとしても、新しいコンポーネント/サービス、依存関係、広範なアクセスの要求などにより、アプリケーションが成熟するにつれて、構成と特権セットが変動する可能性があります。

同時に、サーバーレスでは、より頑強で適切に構造化されたデプロイメント環境を提供します。サービスプロバイダは、呼び出し可能なユニットに伝搬される前に、まずインバウンドトラフィックを処理します。一般的に、サーバーレスはマイクロサービスやVMに比べて柔軟性の低い環境であり、**「脆弱な構成」**による脅威を軽減します。軽減された脅威の例としては、CSPが呼び出し可能なユニットごとに宣言されたイベントの外部の呼び出し可能なユニットにインバウンドトラフィックが到達しないようにしたり、TLSの最新の推奨バージョンを常に使用したりすることが挙げられます。このように、CSP の意見が反映された環境では、攻撃対象が減少し、設定ミスの機会が減少する可能性があります。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

フラグメントの数が増えれば増えるほど、サプライチェーンの脆弱性による課題も大きくなります。適切な処理を行わないと、各フラグメントは潜在的に異なる依存関係のセットを使用する可能性があります。それぞれのフラグメントが異なるリポジトリやビルド/デプロイメントツールを使用する可能性があります。イメージベースのサーバーレスを使用する場合、それぞれが異なるベースイメージに依存する可能性があります。その結果、ワークロードの攻撃対象が増加し、様々な依存関係の増加に伴い、潜在的な脆弱性の数も増加します。これらの脅威は、**「脆弱な依存関係」、「ベースイメージの脆弱性」、「リポジトリとベースイメージレジストリへの不正アクセス」、「ビルド/デプロイツールに対する、または介した攻撃」**として特定されています。

5.4.2 アプリケーションオーナーのデプロイメントフェーズの脅威（参照5.3(B)）

このサブセクションでは、デプロイメントフェーズの脅威に関連するサーバーレスの新規性を強調します。

データインジェクションの側面

サーバーレスでは、呼び出し可能なユニットが処理できる様々なソースからの膨大な範囲のイベントを利用します。使用されるイベントの種類は、そのソースに関わらず、潜在的な**「データインジェクション」**の脅威となります。
サービスプロバイダは、適切なタイプのイベントが適切なソースから送られてくるように（さらには適切な基本イベントフォーマットが与えられているように）制御することによって、特定の保護を提供することはできますが、そのような保護は、攻撃者がインジェクションの欠陥を利用することを防ぐものではありません。攻撃者が制御する可能性があり（例えば、別の脆弱性や設定ミスを利用したり、偽の認証情報を取得したりした後など）、イベントの一部として呼び出し可能なユニットに渡されるすべての情報は、脅威とみなされるべきです。

例えば、呼び出し可能なユニットに対するイベントは、オブジェクトストアにロードされたオブジェクトに関する通知をソースとすることができます。このイベントには、ロードされたオブジェクトに関する情報（犯罪者の管理下にある可能性のある情報）が含まれる場合があります。同様に、呼び出し可能なユニットへのイベントは、犯罪者の管理下にある可能性のあるWebリクエストをソースとすることもできます。呼び出し可能なユニットが、データベースアクセス要求の一部として、イベントによって運ばれた情報を（適切なスクリーニングを行わずに）直接使用したとします。その場合、犯罪者は、巧妙に作成された情報を使ってデータベースを変更したり、機密情報を読み取ったりすることができます。

グローバルコンテキストの側面

マルチテナント性は、サーバーレスでは「設計上（by Design）」取り扱われるとされています。呼び出し可能なユニットの各呼び出しは、特定のイベントを処理するため、特定のテナントに帰属させることができます（イベントがサービス全体に及んでいて、特定のテナントに関連していない場合は、少し無視します）。この「設計上」のマルチテナント性への対応は、呼び出し可能なユニットの呼び出し間で情報が漏洩しない限り、有効です。CSPは同じサーバーレスインスタンスを使用して複数のイベントを連続して処理することがあるため、アプリケーションオーナーのコーダーは、あるイベントによる1つの呼び出しからの情報が、後続のイベントによる後続の呼び出しに漏れないようにする必要があります。

呼び出し可能なユニット／関数がインスタンス化される際には、認証、シークレット/トークンの取得、サポートサービスへの接続とコンテキストの確立、の初期化が必要になる場合があります。このような初

© Copyright 2021, Cloud Security Alliance. All rights reserved.

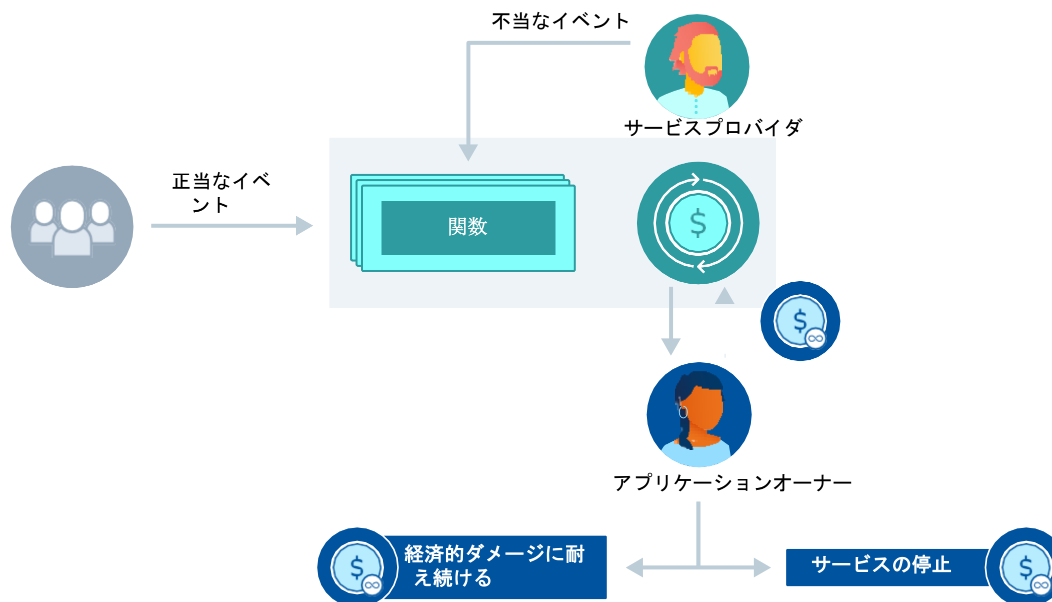
期化は、一連のイベントを処理する一連の呼び出しに対して一度だけ実行する必要があります。呼び出し可能なユニットは、トークン、シークレット、オープンコネクションなどを保持（キャッシュ）するために、グローバルコンテキストを必要とします。このようなグローバルコンテキスは、呼び出し可能なユニットが効率的になるために必要です。さもなくば、たとえイベントが連続して発生したとしても、イベントごとに他のシステムを認証して接続する必要が生じます。

グローバルコンテキストを導入すると、上述したように「**グローバルコンテキストのリーク**」という脅威が発生し、ある呼び出しからの情報が後続の呼び出しに漏れる可能性があります。

コンピュートリソース割り当てのコントロール喪失の側面（「従量課金制」の危険）

VMやマイクロサービスなどの他の選択肢とは異なり、サーバーレスでは、アプリケーションオーナーが使用するコンピュートリソースの量を、セットアップ時に定義された制限値に抑えることはできません。アプリケーションオーナーは、常に使用されるリソースの数をコントロールすることができず、予算や予測を設定し、サービスプロバイダからの請求通知を処理することによって、コントロールを取り戻す必要があります。この新規性のセキュリティ上の側面の1つは、アプリケーションオーナーが、適切な監視と処理を行わないと、上述の**「資金とリソースの枯渇」**という脅威に直面することです。

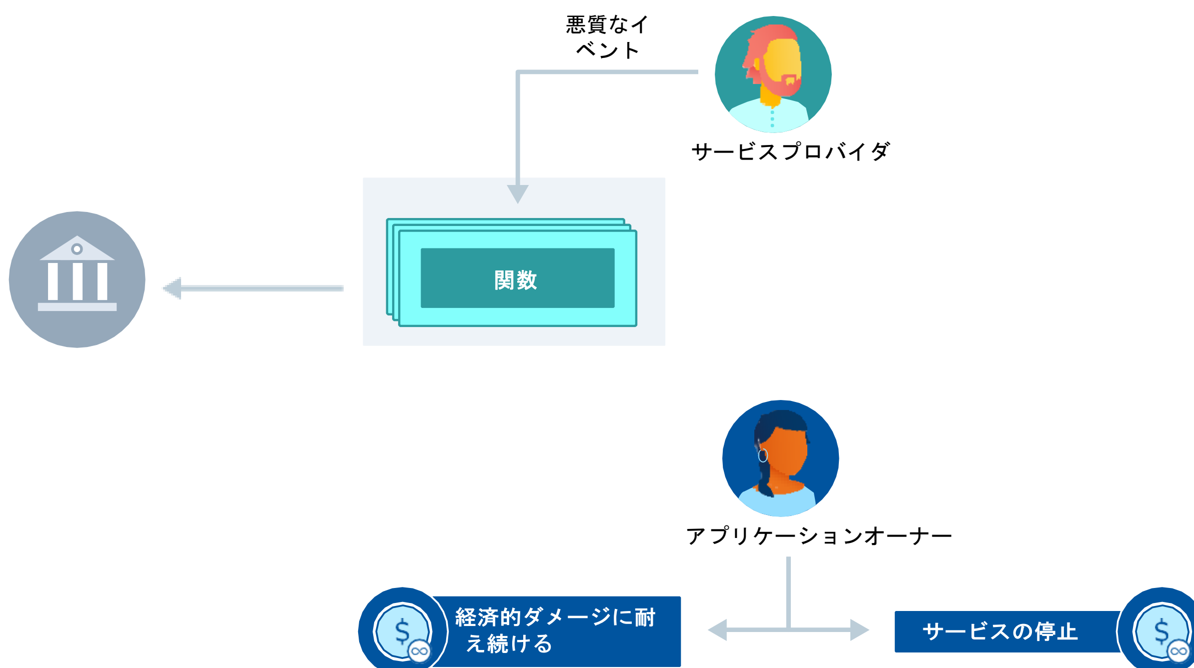
この脅威の一例として、攻撃者がアプリケーションオーナーのワークロードにイベントの負荷を継続的に課すことで、アプリケーションオーナーに*Induced Denial of Service（誘導型サービス拒否）*を引き起こすことが挙げられます。サービスプロバイダが正当なイベントと非正当なイベントの両方を処理するために十分なリソースを提供していると仮定しても、非正当なイベントの継続的な高負荷はアプリケーションオーナーに予期せぬ支出をもたらし、アプリケーションオーナーがサービスの停止や制限を余儀なくされるレベルになる可能性があります。これは、ビジネスの意図やニーズに合わない可能性があります。例えば、*誘導型サービス拒否は*、クラウドのコストを増加させることで直接的に、またはワークロードを停止させることで間接的に、アプリケーションオーナーに金銭的な損害を与えることを目的とした犯罪者による分散型攻撃の一部として実現するかもしれません。



直接的な金銭的損害による誘発的なサービス拒否

© Copyright 2021, Cloud Security Alliance. All rights reserved.

*誘導型サービス拒否*は、**「リソースの豊富さ」**の脅威が顕在化した結果、犯罪者が自分の目的を達成するために呼び出し可能なユニットを利用する方法を見つけた場合にも発生します。例えば、犯罪者が呼び出し可能なユニットを使って自分のコントロール下にある宛先にトラフィックを送信することができるとします。彼は無制限のリソースを、第三者に対するサービス拒否攻撃の一部として使用することができます。サーバーレスでは、呼び出し可能なユニットがアウトバウンドトラフィックを開始することを制限していないことに注意してください。このような攻撃は、サービスプロバイダからの定期的な課金通知を処理しても防ぐことができず、アプリケーションオーナーによる即時の対応が必要となります。このような攻撃が開始されると、アプリケーションオーナーは直ちにそれを止める必要があり、できればエンティティが攻撃される前に、攻撃元をアプリケーションオーナーの呼び出し可能なユニットに特定します。呼び出し可能なユニットの攻撃的な行動を止めるには、サービスを停止する必要があるため、誘導型サービス拒否が発生する可能性があります。



レピュテーションの損傷により誘発されたサービス拒否

第3の例では、犯罪者が呼び出し可能なユニットに自分のコードを実行させることができる場合（サプライチェーン攻撃、またはコードや依存関係の脆弱性の利用など）、呼び出し可能なユニットは便利で豊富なクリプトマイニングのリソースを提供することができます。この脅威は、犯罪者が使用レベルを正規のワークロードの境界内に維持する場合、検出が困難となります。コードを実行すると、犯罪者は、呼び出し可能なユニットの時間制限内に続くイベントによってアクティビティがトリガーされる汎用ボットネットとして呼び出し可能ユニットを利用することもできます。豊富なリソースにより、ボットネットを検知されないようにすることができます。呼び出し可能なユニットは、呼び出し可能なユニットの他のインスタンスをトリガーすることによって終了する前に、ボットネットをさらに持続させることができます。このようなボットネットでは、サーバーレスでアウトバウンドトラフィックが利用できるため、クライアント・サーバー型のコマンド＆コントロールアーキテクチャを使用することができます。
このようなボットネットでは、呼び出し可能なユニットに関連するイベントを再利用することで、修正されたピアツーピアのコマンド＆コントロールを使用することもできます。呼び出し可能なユニットのコードが犯罪者の制御下に置かれると、既存のイベントがボット間に必要な通信チャネルを提供する可能性があることに注意してください。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

したがって、リソース過多は、サーバーレスのアプリケーションオーナーにとって大きな脅威であり、犯罪者に利用されないように適切に対処する必要があります。

フラグメンテーションによる複雑化の側面

サーバーレスアーキテクチャは、断片化されたシステム設計を促進します。このようなアーキテクチャ用に構築されたアプリケーションには、それぞれが特定の目的を持った数十個（あるいは数百個）の異なるサーバーレス関数が含まれている場合があります。これらの関数は、織り交ぜられ、オーケストレーションされて、システム全体のロジックを形成します。サーバーレス関数の中には、パブリックなWeb APIを介して関数を公開するものもあれば、プロセスや他の関数間の「内部接着剤」として機能する場合もあります。関数によっては、クラウドストレージのイベント、NoSQLデータベースのイベント、IoTデバイスの遠隔測定信号、さらにはSMSの通知など、さまざまな種類のソースのイベントを消費することがあります。これらは非常に複雑であるため、犯罪者にとっては潜在的な「脅威」の機会となる可能性があります。これらはすべて、上述の **「壊れた、または安全でない認証」** および **「セキュアでないシークレット管理」** として特定された、認証関連の潜在的な脆弱性の媒介となります。

システムの複雑さに関連するもう1つの例として、アプリケーションオーナーが状況認識能力を失うという脅威があります。制御およびデータパスの一部がサービスプロバイダの制御下にあり（すなわちイベントシステム）、他の部分がアプリケーションオーナー（イベント処理）の制御下にある複雑で断片的なワークロードは、上述の **「不十分で安全ではないロギング/モニタリング」** として示される状況認識の完全な喪失につながる可能性があります。

コードの堅牢性と正当性の側面

サーバーレスは、開発者が実行環境をローカルで完全に再現することができるマイクロサービスやVMとは異なります。サーバーレスベースのワークロードは、データと制御パスの一部をサービスプロバイダにオフロードするため、他のコンピュートオプションよりも多くの開発とテストをクラウド上で行う必要があります。サーバーレスベースのアプリケーションのクラウドネイティブなデバッグオプションは、標準的なアプリケーションのデバッグ機能と比較して、制限されています（さらに複雑です）。
この現実は、サーバーレス機能が、コードをローカルでデバッグする際には利用できないクラウドベースのサービスを利用する場合に特に当てはまります。この脅威は、上述の **「不適切なエラーと例外処理」** として特定されており、不必要に冗長なエラーメッセージや隠れた脆弱性を通じた情報開示として現れる可能性があります。ベストプラクティスや標準的なエラーメッセージのテンプレートがあります。サーバーレスのデプロイメントでは、セキュリティの観点から徹底的にサーバーレスアプリケーションのテストを行い、エラーメッセージが一般的なものであること、データやメタデータを漏らさないことを検証することが重要になります。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

5.4.3 サービスプロバイダのデプロイメント上の脅威（参照：5.3(C)）

この最後のサブセクションでは、サービスプロバイダからの脅威を検討し、サーバーレスに関するそれらの側面について説明します。

隔離の側面

他のサービスと同様に、サービスプロバイダのセキュリティ上の最大の関心事は、クラウドのテナント間で破られない隔離を確立することです。サーバーレスは、クラウドユーザーに提供される隔離に関して、多くの新しい課題をもたらします。イベントシステムとコンピュートシステムの両方が、悪意のあるテナントの攻撃対象となります。これらのシステムの潜在的な脆弱性は、サーバーレスのコアサービスに関連する他の多くの潜在的な危険性とともに、上記で**「クラウドサービスの脆弱性」**として特定されています。

サーバーレスでは、VMやマイクロサービスとは異なり、サービスプロバイダが新たに分離を確立する必要があります。サーバーレスの呼び出し可能なユニットは頻繁に起動と終了を繰り返すため（また、クラウドのユーザーは実際に消費された計算時間に対してのみ料金を支払うため）、インスタンスの起動と終了によるオーバーヘッドを積極的に削減することがサービスプロバイダの利益となります。そのため、サービスプロバイダは、複数のイベントに対応するために既に起動している呼び出し可能なユニットを再利用しています。異なるイベントは、アプリケーションオーナーのワークロードの様々なテナントに関連する可能性があり、呼び出し可能なユニットの呼び出し間の分離がアプリケーションオーナーにより要求され、期待されます。これにより、上述の**「呼び出し可能なユニットの呼び出し間の漏洩」**というサーバーレス特有の脅威が発生します。

2つ目の脅威は**「異なる呼び出し可能なユニット間の漏洩」で、**同じランタイム環境で実行される呼び出し可能なユニット間の隔離問題について考えます。サービスプロバイダによっては、FaaS上の機能の実行が完了すると、後続の機能も同じ実行環境で実装される可能性があります。これにより、犯罪者が以前の呼び出しユニットから残された残留データを収集して実行環境を変更し、後続の呼び出しユニットの脆弱性を悪用する可能性のある、大きな潜在的な攻撃対象領域が発生します。

責任共有の側面

サービスプロバイダのサービスを利用すると、ワークロードのセキュリティに対する責任を共有することになります。しかし、サーバーレスでは、この共有責任を極限まで高めています。VMやマイクロサービスの下では、アプリケーションオーナーとサービスプロバイダの間の責任分担は、コンテナやVMの境界で明確に定義されており、ワークロードの実行は完全にアプリケーションオーナーの制御下にあります（データストアやネットワークなど、サービスプロバイダ側に責任がある場合とは異なります）。サーバーレスの場合、その境界は曖昧です。

ファンクションベースのサーバーレスでは、アプリケーションオーナーのコードは、サービスプロバイダが管理するランタイム内で実行されます。セキュリティは、コードの設計、ランタイムのセキュリティ、および2つの間の相互作用によって決定されます。アプリケーションオーナーのコードは、事前にインストールされたライブラリを含む特定のイメージに依存し、テストされるかもしれません。そのイメージがCSP によって更新されると、コードと新しいイメージの間の相互作用によって、呼び出し可能なユニットの脆弱性が発生する可能性があります。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

さらに、サーバーレスのワークロードでは、機能は、サービスプロバイダによって処理されるイベントによって結合されたアプリケーションオーナーのコードフラグメントによって形成されます。繰り返しになりますが、セキュリティは、コードの設計やイベントシステムのセキュリティだけでなく、両者の相互作用によっても決定されます。例えば、アプリケーションオーナーのコードがイベントシステムを特定の方法で使用する場合です。

これらの例は、サーバーレスにおける責任共有の側面が、VMやマイクロサービスと比較してより複雑であることを明らかにすることを目的としています。この脅威は、上述の**「サーバーレスサービスの正当性」**で特定されています。

6. セキュリティのデザイン、コントロール、ベストプラクティス

マイクロサービスは、ビジネスドメインをサポートする、独立してリリースされ、デプロイ可能なサービスです。サービスは機能をカプセル化し、APIを介して他のサービスにアクセスできるようにすることができます。例えば、あるサービスは会計処理を行い、別のサービスは口座管理取引を行い、さらに別のサービスは報告書を作成しますが、これらのサービスを合わせて、口座管理銀行システム全体を構成することがあります。

サービス指向アーキテクチャは柔軟性があり、独立してデプロイ可能である限り、サービスの境界線をどのように引くべきかを規定しません。技術にとらわれないことは、マイクロサービスの重要な利点の1つです。マイクロサービスの主な特徴は以下の通りです:

1. 独立したデプロイ可能性
2. ビジネスドメインを中心としたモデル化
3. 状態を所有
4. サイズと柔軟性

ドメイン駆動アーキテクチャ(Martin Fowler)に基づき、イベント駆動型マイクロサービス（同期型と非同期型のイベント駆動型マイクロサービス）について多くの議論がなされてきました。同期型のサービスには、依存性のあるスケーリング、ポイントツーポイントの結合、APIの依存性、障害管理、データアクセスの依存性、管理の課題などの欠点があります。イベント駆動型の非同期マイクロサービスは、柔軟性、技術的独立性、ビジネス要件柔軟性、疎結合、継続的デリバリモデルなどの理由で人気があります。

業界はまだ進化していますが、一部のサービスがモノリシックであり、一部のサービスがイベント駆動型のファンクションベースのマイクロサービスであるハイブリッドなアーキテクチャも出てくるでしょう。

FaaSサービスを基本的に利用する、イベント駆動型マイクロサービスのアーキテクチャのベストプラクティスを以下に紹介します：

インタフェース - API＆契約:

- 自動リトライ
- 並行処理
- スケール要件
- メッセージ＆ペイロード
- ユーザーとサービス/セッションの識別
- 失敗、タイムアウト、リトライなどの処理方法、レート制限

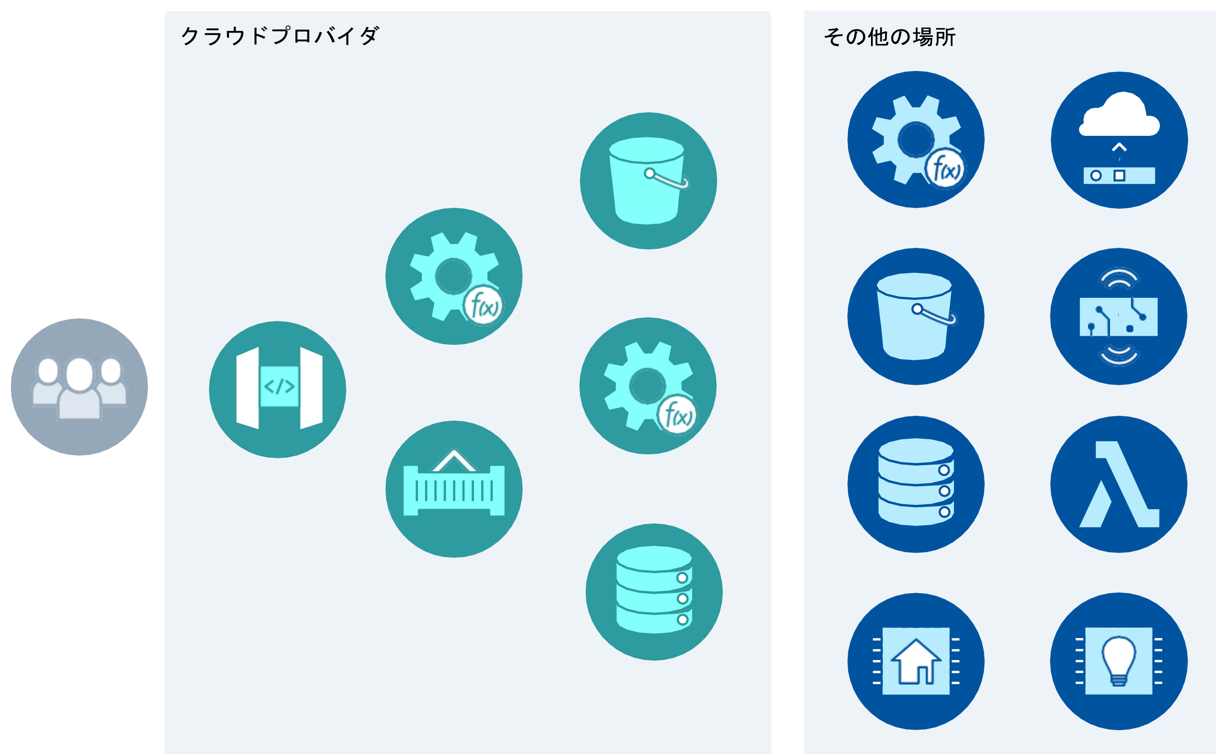
© Copyright 2021, Cloud Security Alliance. All rights reserved.

サーバーレス環境では、セキュリティ対策の進化を余儀なくされる多くのアーキテクチャの変化が起こります。いくつかの例を紹介します。

- **イベント**
  - サーバーレスでは、呼び出し可能なユニットはイベントを介して通信します。これらのイベントは、管理ドメイン間を移動します。したがって、呼び出し可能なユニットは、信頼できない境界を越えて、他の呼び出し可能なユニットにデータを送信する可能性があります。
  - 呼び出し可能なユニットの数は、ワークロードの断片化、呼び出し可能なユニットの寿命の短さ、イベント駆動型アーキテクチャの性質（例：ファイルが変更されるたびに、いくつかの関数が使用される）などにより増加します。
  - 連鎖した関数が使用されており、その中には信頼できる、あるいは信頼できない複数のソースからデータを引き出すものもあります。
- **ネットワーク**
  - ネットワーク構造はサービスプロバイダの管理下にあり、ネットワークログにアクセスできるのはサービスプロバイダだけです。
  - ネットワーキングの隘路（サーバーレスプラットフォームのパフォーマンス問題など）は、サービスプロバイダの管理下にあります。
  - 新しい境界、あらゆるネットワークセキュリティコントロールは、ゼロトラストモデルとアイデンティティコントロールを用いて再編成するか置き換える必要があります。
  - 転送中のネットワークセキュリティは、主にサービスプロバイダが責任を負います。利用者は、アプリケーションの転送セキュリティを設定する必要があります。
- **ライフサイクル**
  - ライフサイクルは全く異なります。関数は破壊されるまでの期間がミリ秒単位であり（限定的なライフサイクル）、セキュリティ監視システムに影響を与える可能性があります。
  - 呼び出し可能なユニットが頻繁に起動や停止するため、寿命が短いとIAMやシークレット管理システムに影響を与える可能性があります。
  - 制御および処理する関数/イメージの数が非常に多いため、完全性を確保するために必要な制御に影響を与えます。
- **攻撃対象領域（第5章で詳細に説明）**
  - サーバーレスが提供するインフラストラクチャの抽象化により、従来の攻撃対象領域は減少しています。
  - クラウドサービスのインスタンス数が大幅に増加し、それに伴い、設定ミスの機会、適切なIAMロールの設定の複雑さ、アプリケーション開発ライフサイクルにおけるすべてのワークロードに対する適切なセキュリティプロセスの設定が必要になります。
- **モニタリング**
  - 従来、アプリケーション（Nginx/Apacheなど）の内部で結合されていたロギングやモニタリングのような側面は、現在ではユーザーから抽象化されています。そのため、コードに沿って適切なロギングやモニタリングのイベントを追加するために、設計の見直しや開発者への新たな要求が必要になります。
- **セキュリティ対策**
  - ファイアウォール、IDS/IPS、SIEMなどの従来のセキュリティ対策では、接続性と統合性が高まるこの新しいパラダイムに効率的に対処できません。そのため、サーバーレスコンポーネントのセキュリティを確保するためには、より分散されたセキュリティアーキテクチャが必要となります。
  - サーバーレスのセキュリティに対する新しいアプローチが登場しており、アプリケーションオーナーが検討する必要があります。このトピックについては、本稿の*「サーバーレスセキュリティの未来像」の*章で詳しく説明します。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

下の図は、サーバーレスアーキテクチャの図で、脱ペリメタ化が顕著に表されています。



サーバーレスセキュリティの視点

上の図は、アプリケーションを構成するサービスの集合体（疎結合）を表しています。これらのサービスは、静的なファイアウォールなどの従来のセキュリティの信頼の境界を必ずしも持たないものです。

サーバーレスのパラダイムでは、関数を使用し、サードパーティのサービス（ホスティングなど）と正しく組み合わせることで、組織が従来のインフラストラクチャの管理や従来のセキュリティの他の側面を気にする必要なく、エンドツーエンドのアプリケーションを実行することができます。

サーバーレスでは、適切なセキュリティパターンがまだ登場していないため、アプリケーションオーナーにはより慎重なセキュリティアプローチが求められます。それらのパターンは、この分野の論文や他の論文を参考にしながら、時間をかけて成熟させていくことになるでしょう。

6.1 サーバーレスの設計上の注意点

アプリケーションアーキテクトは、サーバーレス技術に存在する固有の弱点と、設計で導入できる弱点の両方を認識しておく必要があります。これらの弱点を理解することで、実装する必要のあるセキュリティ対策をよりよく把握することができます（これらの対策については次の章で説明します）。

サーバーレスアーキテクチャには、セキュリティの観点からも多くのメリットがあり、セキュアなマイクロサービス設計の検討事項の一部として重きを置かれています。それらのメリットの一部を紹介します。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

1. ステートレスでエフェメラル。短命なサーバーレス関数は、暗号化されていないデータを短時間だけメモリ上で処理します。サーバーレス関数は、ローカルディスクへの書き込みを行いません。したがって、状態を持続させる必要がある関数は、外部のデータストアに依存するため、*長期的なターゲットを想定して設計された攻撃に悪用される可能性が低くなります。*
2. 各サーバーレス関数は、ミクロに焦点を当てたサービスを実行するために、データのサブセットのみを必要とします。*そのため、この関数が必要なデータのみにアクセスするための正しい権限を持っている限り、ある関数の悪用に成功しても、どのデータが流出する可能性があるのか焦点を絞ることができるはずです。*
3. サーバーレスアプリケーションは、CSPが管理するコンテナ内、または自己管理のコンテナ内で実行されます。そのため、イミュータブルなコンテナイメージ上で動作するコンテナに固有のセキュリティ上の利点があります。長寿命のサーバーを必要としないコンテナは、コンテナイメージやコンピュートインスタンスに継続的にパッチを当てることができます。脆弱性の*ある、あるいはパッチが適用されていない基盤上で動作することへの懸念が軽減されます。*

アーキテクトや開発者が考慮すべきその他の検討事項は以下の通りです：

ベンダーロックイン：CSP環境では様々なサービスや依存関係を統合しているため、関数は、他のサービスプロバイダの環境と互換性がないクラウドサービスを使用している可能性があります。例えば、バックエンドデータベースが RDS や MS SQL Azure などにある場合があります。

デプロイツールの制限と実行環境の制限：アプリケーションのニーズに基づいて、必要なツールや実行環境を理解し、それらのニーズに基づいてプロバイダを選択することが重要です。また、サービスの全体的な機能に影響を与える可能性のある制限事項を調査することも重要です。

6.1.1 サーバーレスプラットフォームのデザインがサーバーレス・マイクロサービス・セキュリティに与える影響

先に紹介したように、関数/アプリケーションオーナーは、アプリケーションの管理・制御に対するより大きな責任を負っています。一方でプラットフォームのインフラストラクチャやセキュリティに対する関心は低く、サービスプロバイダの責任範囲となります。このような責任の移行に伴い、機能/アプリケーションオーナーは、インフラストラクチャに対する可視性と責任を失うことになります。さて、関数のオーナー側からは、次のような疑問が出てきます。

- 関数はどこで動いているのか？
- その関数の実際のネットワーク露出度は？
- アイドル状態のコンテナで実行される関数は、前の実行ですでに初期化されているものを「ウォームスタート」、新たにインスタンス化されたコンテナを「コールドスタート」と呼ぶのか？
- キャッシュメモリに以前のデータがまだ残っているか？

まずは、サーバーレス・マイクロサービスの設計プロセスで必要となる、固有の設計特性を理解することから始めましょう。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

関数には本来、パブリックな場に面した出口があります。

サーバーレスの世界では、CSPはクラウドカスタマと比較して、セキュリティに対してより多くの責任を負います。クラウドカスタマ（CSC）の責任は限定的です。例えば、CSCは自分のアプリケーション内のセキュリティや、サーバーレスのコンポーネントやサービスの安全な構成に責任を負います。CSPがデフォルトで提供するセキュリティは、すべての利用者が必要とするすべてのセキュリティ要件を満たしていない、または対応していない可能性があります（例えば、コンプライアンスのため）。CSPが提供するAPIゲートウェイは、パブリックなインターネット上のどこからでもアクセスでき、APIキーによってアクセスが制御されます。顧客はビジネスニーズに基づいて、サポートするコントロール（トランスポートベースのアクセスコントロール）を設定する責任を負うことになります。

データのインターネットへの公開を制限するために、以下のコントロールを使用してセキュリティを強化するように設計します：

i. ネットワークポリシーを適用し、各CSPが提供するVPC（Virtual Private Cloud）ネットワークポリシー設定を使用して、そのFaaSから到達可能なエンドポイントを制限します。
ii. データストア/サービスにアクセスできるエンドポイントを制限できるサービスポリシーやリソースポリシーを適用します。それによりデータの流出経路を減らすことができます。

例えば、AWS VPCエンドポイント、Azure VNet Serviceエンドポイント、Google VPC Service Controlsなどがあります。

設計上、大規模なアプリケーションをマイクロサービスに分割すると、サーバーレスで展開されたマイクロサービスのアーキテクチャを完全に可視化することが難しくなります。

サーバーレスアプリケーションは、マイクロサービスアーキテクチャに基づいて構築されており、論理的にフォーカスされた多くの機能を構築して同時実行し、処理をスケールアウトすることができます。

しかし、単一のアプリケーションであれ、より大きな機能を持つマイクロサービスの分散型APIであれ、利用可能な機能の数を増やしていくと、その機能がどのように実行されるかを完全に可視化する中で課題が生じます。例えば、次のような質問は検討に役立ちます：

1. これらの機能はすべてVPC内で実行されているのでしょうか？特にデータの重要性のために公的な公開を最小限に抑える必要がある場合はどうでしょうか？
2. 各機能に作成したすべての役割は、必要最小限の権限を許可するように調整されていますか？
3. 開発者は既存のロールを再利用し、データを読み取る必要のある関数と、データの値を処理・更新する必要のある関数が同じロールを使用するようにしたのでしょうか？
4. アプリケーションの再設計で、HTTPイベントではなくイベントキューで関数を起動することが指定されたとします。この場合、HTTPイベントトリガーはデプロイオプションの一部として削除されましたか？
5. HTTPイベントで起動できる機能は、すべてAPIゲートウェイの背後にあるのでしょうか？

サーバーレス環境に関する具体的な懸念事項としては、以下のようなものがあります。

A. 不十分な関数のロギングとモニタリング
B. セキュアでないサーバーレスのデプロイ設定
C. アプリケーションシークレットのセキュアでない保存
D. サードパーティの依存関係のセキュアでない管理

© Copyright 2021, Cloud Security Alliance. All rights reserved.

A. 不十分な関数のロギングとモニタリング：イベント駆動型マイクロサービス/関数の実行環境を詳細に可視化するためのコントロールは以下の通りです：

i. 不適切な関数ロギング：ロギングは、所定の期間にエンティティによって実行された活動の記録を提供します。これにより、システム/アプリケーションに対する洞察が得られ、インシデントが発生した場合のデバッグ、評価、リプレイの機能が提供されます。サーバーレスにおけるロギングのベストプラクティスは、ログの値を構造化することです。構造化されたログは解析しやすくなります。もう一つのベストプラクティスは、必要なログレベルや冗長性を決定し、必要なときにログが価値を提供できるようにすることです。ログは、異常なパターンや状態を監視し、是正措置を講じるために業務に警告・通知するために活用されるべきです。アプリケーションのパフォーマンスとセキュリティの全体的な監視を容易にするために、統合されたログを使用して一元化することをお勧めします。
プラットフォームプロバイダのロギングは、関数の実行量、実行時間、メモリ使用量などの統計を収集するのに役立ちます。アプリケーションのエラーの可視性は、サーバーレス関数内に必要に応じてロギングステートメントを追加することで、自分でコントロールできます。例えば、あなたのエラーログは、障害が自分で定義したプロセス内で発生したのか、予期しないデータ入力から発生したのか、あるいはサードパーティの関数プロセスの結果なのかを特定することができるでしょうか？

ii. 不適切な関数のモニタリング：アプリケーション監視ツールやセキュリティ監視ツールを使用して、関数の実行頻度や論理的な実行経路を可視化します。自分が公開しているすべてのAPIやエンドポイント、下流または依存するすべてのAPIを監視し、発見する必要があります。また、どのイベントやロールが機能を実行しているか、予期せぬ実行パスやメソッドがないかを監視し、発見する必要があります。

B. セキュアで**ないサーバーレスのデプロイ設定**：プラットフォームプロバイダのデフォルト設定は、特定のマイクロサービスのユースケースに必要なセキュリティレベルに影響を与えます。

サーバーレスでは、特定のニーズやタスク（ネットワークポリシー、コマンドラインインターフェース）に合わせたカスタマイズや構成設定が可能です。

サーバーレスのセキュリティは、サーバーレスのマイクロサービスと統合する機能やいくつかの上流と下流のサービスの設定に依存します。それらのPaaSサービスの設定と、セキュリティ強化のオプションをすべて理解する必要があります。重要な構成設定を誤る確率は、非常に大きな影響を与える可能性があります。

例えば、PaaSのデータサービス（例：Amazon S3、EMR、RedShift）を使用している場合でも、デフォルトの設定を使用しており、その設定によってデータが公開される可能性があります。関数がデータの読み取りと処理だけを必要とする場合に、関数がデータの読み取りと書き込みを許可するデフォルトのプラットフォームプロバイダのIAMロールを使用していないでしょうか。

ベストプラクティスの提案：

i. セキュリティ設定とポリシーは、マイクロサービスのユースケースに基づいて定義し、プラットフォームプロバイダのデフォルト設定への依存度を最小限に抑える必要があります。マイクロサービスおよびデータのセキュリティ要件とリスクポスチャに基づいて、各サービスをセキュアに設定する方法を決定します。

ii. サーバーレスでは、特定のニーズやタスク（ネットワークポリシー、コマンドラインインターフェース）のためのカスタマイズと構成設定を行います。重要な構成設定を誤る可能性は、サーバ

© Copyright 2021, Cloud Security Alliance. All rights reserved.

ーレスに大きな影響を与えます。そのため、マイクロサービスに対してセキュリティテストを実施し、設定がすべてのセキュリティ要件を満たしていることを確認する必要があります。セキュリティテストは、関数やデータにアクセスできるメソッドや役割を検証するのに役立ちます。さらに重要なことは、アプリケーションセキュリティテストによって、アプリケーションロジックや入力の検証が、インジェクション攻撃やその他の構成上の弱点によってまだ攻撃されていないことを明らかにすることです。

C. アプリケーションシークレットのセキュアでない保存

アプリケーションが大規模化・複雑化するにつれ、アプリケーションシークレット（APIキー、暗号化キーなど）を保存・管理することが重要になってきます。

既製の CSP サービスでは、すべてのテナントのセキュリティニーズに対応できません。テナントは、自社のセキュリティのレベル（PIIデータ、機微データ、コンプライアンス要件など）を判断し、デフォルトのCSPサービスに加えてセキュリティをアップグレードする必要があります。機微データや規制の厳しいデータにアクセスして処理するサーバーレスアプリケーションを設計していますか？そうであれば、そのデータをどのように保護するかというセキュリティ要件が高まり、外部の攻撃者によるデータ流出だけでなく、悪意のあるインサイダーも考慮する必要があります。

プラットフォームプロバイダは、デフォルトでは暗号化されていないコンピュートインスタンス上でサーバーレスアプリケーションを実行しています。さらに、サーバーレスアプリケーションは、ほとんどの場合、デフォルトで暗号化されていないプラットフォームプロバイダによって提供されるPaaSデータサービスに依存しています。すぐに使用できる CSP は、すべてのテナントのセキュリティニーズに対応しているわけではありません。テナントは、自社のセキュリティレベル（PIIデータ、機微データ、コンプライアンス要件など）を判断し、デフォルトのCSPサービスに加えてセキュリティをアップグレードする必要があります。

ベストプラクティスの提案：

i. プラットフォームプロバイダが、サーバーレスコードを機密性の高いコンピュートインスタンス上で実行するようにデプロイするオプションを提供しているかどうかを確認します[IEEE Spectrum, 2020]。これは、アプリケーションとデータのセキュリティ要件とリスクポスチャを十分に満たすために、関数を使用するか、他のコンピュートオプションを使用するかを決定するのに役立つかもしれません。
ii. サーバーレスアプリケーションに組み込まれた各PaaSデータサービスのデフォルトまたはマネージドな暗号化オプションを確認し、機密データや規制の厳しいデータを保持します。PaaSデータサービスの使用と構成が、アプリケーションとデータのセキュリティ要件を満たすことを確認します。満たされない場合は、アプリケーション層の暗号化を補償コントロールとして実装します。
また、コンフィデンシャルコンピュートは高価であり、まだ進化中のため、ソリューションの実現可能性や、コンフィデンシャルコンピュートがサービスの実行に与えるパフォーマンスへの影響を考慮して設計を決定する必要があります。

D. サードパーティの依存関係のセキュアでない管理：サーバーレスアプリケーションのサードパーティ製ライブラリへの依存

時には、サーバーレス機能はサードパーティのソフトウェアパッケージやオープンソースのライブラリに依存し、さらにはAPIコールを通じてサードパーティのリモートWebサービスを利用します。サーバーレスアプリケーションのコードやデプロイの脆弱性スキャンでは、依存しているサードパーティのライブラリの可視性は限られています。特に問題となるのは、ソースコードリポジトリの外で管理されているサードパーティのライブラリにもたらされた脆弱性です。サードパーティの依存関係には、サーバーレスアプリ

© Copyright 2021, Cloud Security Alliance. All rights reserved.

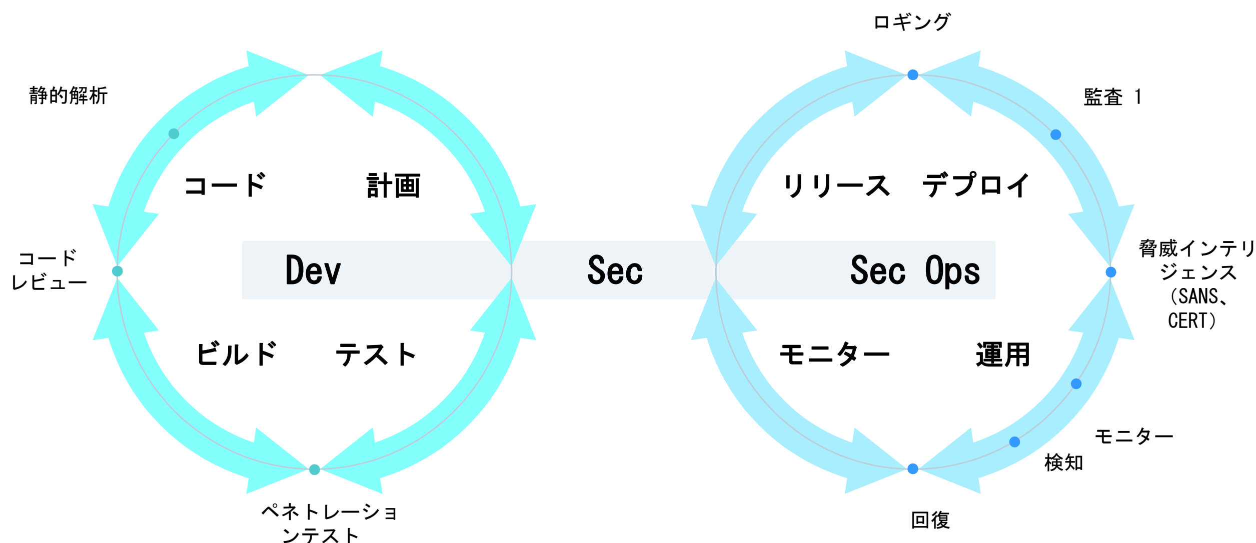
ケーションを攻撃にさらす可能性のある脆弱性が存在する可能性があるため、注意が必要です。

ベストプラクティスの提案：

i. サードパーティ製ライブラリのソース構成分析をデプロイに組み込みます。これにより、依存関係の範囲だけでなく、すでに知られている脆弱性のあるコンポーネントでリスクをもたらしていないかどうかを確認することができます。
ii. セキュリティモニタリングソリューションを使用して、実行時に脆弱なサードパーティライブラリを特定し、含まれているが使用されていないライブラリを特定します。冗長なライブラリを削除することで、脆弱性が不必要に含まれるリスクを低減します。

このセクションでは、サーバーレスのセキュリティを確保するためのセキュリティコントロールとベストプラクティスについて簡単に説明します。

クラウドサービスではCSPとCSCの間の責任共有モデルに従いますが、これはサーバーレスでも同じです。サーバーレスが様々な組織で採用されるようになると、ファンクションやイメージベースのモデルにおけるプラットフォームサービスプロバイダの責任と利用者の責任をそれぞれ理解することが重要になります。サーバーレスアーキテクチャでは、CSPがクラウドのセキュリティ責任（サーバーとオペレーティングシステムのセキュリティ）を負い、テナントはIAAA（AuthN、AuthZ、監査ログ）、SDLC（コードレビュー、静的解析、ビルド、テスト、リリース、デプロイ）、データ保護（静止時、移動時）、ポリシーの実施（例：コードレビューは2人のピアによって行われなければならない、コードはクリーンな脆弱性スキャンの後にのみリリースされるべきである）などのセキュリティ管理によってクラウド内のセキュリティに責任を負うことになります。



6.2 FaaSのコントロール

注：FaaSは上位レイヤーであり、コンテナイメージベースのサーバーレスの上に構築することができるため、FaaSに適用されるすべてのコントロールは、コンテナイメージベースのサーバーレスにも適用されます。（このセクションでは主に、企業が独自のプライベートFaaSを構築することを選択した場合に実装すべきコントロールについて説明します）。

現在、私たちはFaaSモデルに移行しており、いくつかのオープンバージョンのFaaSがあります。アプリケーションチームのセキュリティ担当者は、機能を実行するコンテナにエクステンション（Lambda Extensions、Knativeなど）やエージェントを追加できるようになりました - ビジネスロジックレベルの問題を解決するには、ファンクションそのものが最適です。
コントロールは様々な方法で見ることができます：

a. プラットフォームの構成監査（コードレビュー、SAST、DAST、ポリシーの実施）をCI-CDパイプラインの一部として実施。
b. プラットフォームの構成要素と脆弱性（サーバーレスプラットフォームのセキュリティのほとんどをクラウドプロバイダが担っています。それは、ホストオペレーティングシステム、コンテナ、オーケストレーションサービス、サービスメッシュなどです）に加え、ネットワークポリシー違反の検出、プラットフォームレイヤーの脅威の検出と管理を行います。つまり、企業がプライベートFaaSの導入を決定した場合、すべてのレイヤーのセキュリティ管理を構築する必要があります。
c. ファンクション構成監査（コードレビュー、SAST、DAST、ポリシーの実施）は、CI-CDコントロールとランタイムコントロールの下に再びバケット化することができます。
d. ファンクションコンポーネントと脆弱性（サーバーレスの脆弱性の多くはプログラミングに関するもので、インジェクション、認証の失敗、アクセスロールの悪用、既知の脆弱性を持ったままのデプロイ、セキュアでないシークレットの保存、セキュアでないデプロイ設定、不適切な例外処理、不十分なログや監視など、OWASP のセキュアコーディングベストプラクティスに該当するものです。
e. アイデンティティおよびアクセス管理（ユーザーおよびアプリケーションのアイデンティティ管理、フェデレーション、SAML 2.0、アクセス制御、多要素認証）
f. ファンクションワークロードのセキュリティ- システムの完全性監視、アプリの許可リスト、アプリのセキュリティ強化、アンチマルウェア、エクスプロイトの防止・検知・対応。

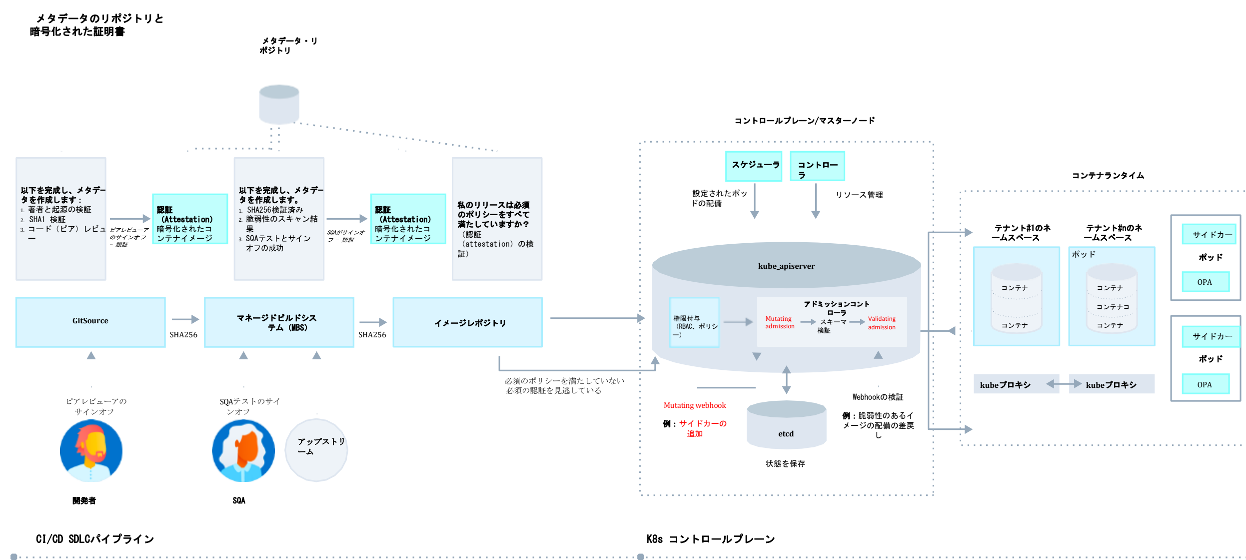
繰り返しになりますが、FaaSのコントロールカテゴリを定義する必要がある場合、次のようにハイレベルなカテゴリにバケット化することができます。

a. プラットフォームサービスプロバイダのAPIおよび管理コントロールと統合
b. CI-CDパイプラインのセキュリティ管理には、コンポーネント、コンフィグ、脆弱性スキャンなどがあります。
c. アイデンティティとアクセス管理
d. プラットフォーム層での検出とランタイムでの検出制御により、ポリシー違反、失敗したファンクション/トリガーなどを検出します。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

6.3 CI-CDパイプライン、ファンクションコード、コードスキャン、ファンクションとコンテナのポリシーの実施

	プランニング	プランニング	プランニング	プランニング	プランニング	プランニング	プランニング	プランニング
セキュリティコントロール	• システム設計 • ビジネスプロセス • ポリシー、手順 • 実用最小限の製品（MVP） • リスクアセスメント • 要件の収集	• アプリケーション開発 • コード、設定、ポリシー、YAMLを保存するソースコードリポジトリ • ユニットテスト • コードレビュー • 静的解析/SAST セキュアコーディング	• コンテナイメージ • プライベートレジストリ • ビルド管理 • CI/CD管理 • 構成	• 統合テスト • 動的分析セキュリティテスト（DAST） • 受入テスト • ペネトレーションテスト	• リリースパッケージのためのチェックサム電子署名の生成 • バイナリ、ベースコンテナイメージ、設定、スクリプト • MVPベースライン • CI/CDオーケストレーター • アーティファクトリポジトリ	• 仮想化エンジン • K8sプラットフォームのライフサイクル • サービスメッシュ • アイデンティティとアクセス管理 • プラットフォームデータ • 配備ポリシー • 構成 • ETCD	• クラスタ管理 • ロードバランシング • システムのスケーリング • レートリミッター • バックアップ管理 • 操作ダッシュボード • コンテナの分離 • アプリケーションのアクセスとデータ • 観測性 • コンテナのイメージスキャン	• ユーザー、ネットワーク、アプリケーション、データのログイベント • ログ解析監査 • 運用監視 • アラートと通知 • ログの集約とアーカイブ
脆弱性	• 安全ではない無防備なデータキャッシング • セキュアな設計 • APIゲートウェイによるAuthN, AuthZの導入	• データインジェクション • OWASPトップ10 • 不適切な例外処理	• 不安定/脆弱な構成 • API OWASPトップ10	• サードパーティを含むライブラリの依存関係	• 不安定/脆弱な構成 • API OWASPトップ10 • 自動化された配備作業に対する攻撃 • 悪用された画像レポジトリ	• 広く一般的な許可 • 認証の失敗 • シークレットやトラフィック（保存時、移動時）の安全でない管理	• 脆弱なイメージ • 広く一般的な許可 • 破られた認証API OWASPトップ10 • ポータルの脆弱性	• ロギングとモニタリングが不十分 • マルウェアの埋め込み



すべてのコード変更は、ピアレビュー、マニュアルコードレビュー、または自動化（静的分析）を経て、リリースに反映されます。すべての変更/更新は、pullリクエストによってサブミットされ、変更を行った人以外の人がコードレビューを行います。レビューを受けた変更は、マスターブランチにマージされ、テストされた後、本番環境にデプロイされます。

静的スキャン

アプリケーションの開発やデプロイが高速化する中で、デプロイ前に「ファンクションやコンテナのコード」をスキャンする自動化された方法が重要になります。静的分析ツールは、CI/CDパイプラインに組み込まれたソースコードの問題点を分析します。特定の種類の問題が伝播するのを防ぐゲートを持つことが不可欠になります。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

スキャンを成功させるためには、以下のことが必要です：

- 環境全体の深く明確な可視化 – スキャン対象となる資産のディスカバリとコントロールをシンプル、簡単、直感的に行うことができます。
  - これは、コードがチェックインされると同時に、コードベース全体ではなく、影響を受けたコードや変更されたコードだけをスキャンする機能を意味します。
  - 他のバージョンのコードを含めて、すべてのコードに依存するライブラリ、バイナリ、およびコードの変更を調べることができます。
  - 構成上の問題点を探すために、ファンクションベースの サーバーレスとコンテナイメージベースの サーバーレスのデプロイマニフェストをスキャンします。
- 結果の精度 – 見つかった問題のノイズを除去するために、フォールスポジティブとフォールスネガティブをコントロールする機能が必要です。
- タイムアウトの検証 – ファンクションには制限された時間があり、時間を超えると停止してしまうため、問題が発生する可能性があります。静的スキャンコントロールは、コードが突然無効化(prempted)されるのを防ぐ可能性のあるコード部分など、コードの問題を特定して修正する必要があります。
- 開発環境やIDE（Visual Studio、Eclipse、IntelliJなど）との統合が容易であること。開発者はツールを使ってファンクションを開発しますが、統合が容易であれば、開発者はコードの問題点を早期に警告し、修正することができます。
- 新たな問題が発見された場合、**早期のスキャンと定期的またはオンデマンドでの再スキャン**をサポートします。コードのコンポーネントに含まれる新しい脆弱性が定期的に発見されるため、これは不可欠です。
- オープンソースの問題 – オープンソースコンポーネントと、アカウント内のすべてのオープンソースライブラリのインベントリをコードスキャンする必要があります。これにより、必要に応じて問題を修正することができます。
- プログラミング言語の幅広いサポート – コンパイル言語とインタプリタ言語の両方をサポートする必要があります。ファンクションの開発者はさまざまな言語で作業を行うため、静的スキャンにはさまざまなバインディングが不可欠です。
- ソースコードやバイナリへのアクセス – git、svn、GitHub、bitbucketなどのソースコードツールとの連携
- Jenkins、TeamCity、Bamboo Maven、Antなどのツールを使ったSDLCや**CI/CDパイプライン**での統合の簡便さ。ソフトウェアの動的バインディングを確認するのに最適な場所です。
- JiraやServiceNowなどのバグトラッキングシステムと簡単に統合できます。これにより、コードをスキャンした際の結果を簡単に統合することができます。
- バッファオーバーフロー、SQLインジェクション、セキュアでないデシリアライゼーションなど、ツールが実行するチェックの種類を制御することができます。

コンフィグテンプレートスキャン

ファンクションをデプロイする際には、自動的にではなく、IaC（Infrastructure-as-Code）テンプレートを使用します。CloudFormationやTerraformなどのテンプレートをスキャンすることで、設定上の問題なくファンクションが安全にデプロイされているかどうかを確認することができ、結果としてセキュリティが損なわれることはありません。

また、テンプレートでは、ファンクションの入出力、ファンクションのつなぎ方、開いているポートなどを確認し、既知の不正な動作と比較して、問題の修正を支援します。入力がインターネットに公開されている場合、コードに厳格なチェックを加え、リスクを強調する必要があります。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

ポリシーの実施

コードテンプレートのスキャンは、サーバーレスのセキュリティにとって重要です。コードスキャンと関連するポリシーは、異なるステージで実施することができます。このような場合、2つのワークフローがあります：

- CI/CDパイプラインにポリシーを実施することは、開発者がコードスキャンツールを簡単に使えるようにすることに成功します。このことは、開発者が問題点を素早く見つけて修正することに役立ちます。しかしながら、多くの場合、セキュリティチームはCI/CDパイプラインを知らず、コントロールすることもできません。-- これは良い第一歩ですが、デプロイされたすべてのファンクションがスキャンされることを保証するものではありません。
- デプロイコントローラ - デプロイコントローラとは、CloudFormationコントローラ、Knativeコントローラ、Terraformコントローラのことです。プッシュされる必要のあるコードは、すべてこのコントローラを経由します。セキュリティチームは、この場所でポリシーを実施することができます。CI/CDパイプラインがどこを通っているかに関わらず、すべてのファンクションをチェックすることができます。

ポリシーには、実施モード（すべてのユーザーの機能へのアクセスを禁止するポリシーなど）があり、ポリシーに適合しないものがデプロイされるのを防ぎます。監視モードでは、アラートが発生しますが、ファンクションの実行は可能です。これにより、インシデントが発生し、JiraなどのCI/CDツールでチケットが発行されることになります。

開発環境、テスト環境、プリプロダクション環境、本番環境など、さまざまな環境に応じて異なるポリシーが必要となります。本番環境では、他の環境に比べてより厳格な管理を行い、他の環境とは明確に分離する必要があります。

ファンクションおよびファンクションにアクセスするサービスに対するアイデンティティおよびアクセス管理の制御を適用することができます。

セキュリティチームがサーバーレスのファンクションをコントロールできる範囲は限られているため、最も重要なことの一つは**サーバーレスの**セキュリティは、IAMのパーミッションを理解して管理することです。IAMには2つの部分があります。

a. ファンクションの作成、デプロイ、削除に使用されるユーザーまたはサービス/ロールのIAM
b. デプロイされたファンクション/コンテナサービス自体のIAMロール/パーミッション

IAM権限の管理は、新しいファンクションが作成、削除、デプロイされるような、非常にダイナミックで変化の激しい環境では難しくなります。以下のことを行う必要があります：

a. **FaaS**のセキュリティの第一歩は、IAMロールとパーミッション（変化のスピードが速いため）、各ファンクションが持っているパーミッションと、ファンクションのデプロイに必要なパーミッションを可視化することです。見えないものをコントロールすることはできません。ファンクションは、セキュリティにおける最小権限モデルを使用する必要があります。
b. 次のステップは、IAMパーミッションのガードレールを作成することで、これは環境内でファンクションに対して許可されるパーミッションのセットになります。デプロイされたファンクションはすべて、そのパーミッションで設定されるべきです。それよりも高いIAMパーミッションを持つファンクションは防止されるべきで、例外パスを通して扱われます。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

前述の「**ポリシーの実施**」の項で定義したように、ビジネスロジックがデプロイされる際にデプロイパイプラインにコントロールを配置したり、ファンクションをデプロイするコントローラにコントロールを配置したりすることで、機能を実現することができます。

ゲートウェイとインタフェースの制御

コード内の外部インタフェース（API）は攻撃対象領域を増やします。FaaSやコンテナイメージベースのサーバーレス環境への外部からのアクセスは、すべてゲートウェイを介して行われます。このため、環境のセキュリティにはゲートウェイが重要となります。

外部からのアクセスを保護するために、ゲートウェイは「OWASP トップ 10」のリスクに対するコントロールと改善をサポートする必要があります。*詳細は「6.3.7.2 API セキュリティ（OWASP トップ 10）」を参照のこと。*

データ保護、および暗号化/KMSサービスとの統合により、保存中および転送中のデータを保護します。

データ保護規制、特に規制の厳しい業界（GDPR）では、FaaSは暗号化とKMSを活用して、保存時と転送時のデータを保護することができます。責任共有モデルに注目すると、データとその説明責任は利用者の責任です。

ファンクションオーケストレーションのコントロールと検証

「ステップファンクション」は、ファンクションの連鎖を統合管理するのに役立ちます。1つのステップが失敗した場合、連鎖全体が失敗することを保証するために、セキュリティコントロールが必要です。

検出と対応

- 故障検出
- ポリシー違反の検出
- 脅威/侵害の検出
- 応答の自動化
- ランタイム検出とポリシーの実施

CSPの機能は大きく異なり、サーバーレスで障害検知を行うサードパーティベンダーもあります。一つの良い方法は、CSPのネイティブな機能を評価し、他にサードパーティの世界から何が付加価値をもたらすかを確認することです。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

6.4 コンテナイメージベースのサーバーレスのための差分/追加コントロール

クラウドネイティブなコンピューティングは、非常に複雑で、また継続的に進化しています。コンピュートリソースの利用を実現するコアコンポーネントがなければ、組織はワークロードの安全性を確保できません。開発者は、マルチテナントのアプリケーションを共有されたホスト上に展開し、必要に応じて個々のコンテナを起動したり停止したりすることができるため、コンテナを利用することで、サーバーを簡単に最大限に活用することができます。このようなニーズに対応するためには、開発者はコンテナを実行するための適切な環境が必要です。

コンテナはソフトウェアベースの仮想化を提供するため、他のサービスを無効にした読み取り専用のOSであるコンテナ専用OSを使用することが重要であり、これにより攻撃対象を減らすことができます。また、隔離とリソースの制限を行うことで、開発者はサンドボックス化されたアプリケーションを共有のホストカーネル上で実行することができます。

テナントが独自のOSイメージを持ち込むことができるコンテナイメージベースのサーバーレスプラットフォームや、プライベートなコンテナイメージベースのサーバーレスプラットフォームでは、Seccompを使用して、OSがセキュアであること、セキュリティ構成に従うこと、最小限のOS構成を使用すること、システムコールが悪用されないことを保証することが適切です。Seccompは、secure computing modeの略で、Linuxカーネルのバージョン2.6.12から搭載されている機能です。Seccompは、プロセスの権限をサンドボックス化し、ユーザー空間からカーネルへの呼び出しを制限するために使用されます。
オーケストレーションエンジンでは、ノードに読み込まれたseccompプロファイルをそれぞれのポッドやコンテナに自動的に適用することができます。同様に、Windowsでは、Syscallのサンドボックス化、Windowsのプロセスごとのカーネル隔離のフィルタリング、Win32kのプロセスごとのターゲットの制限とブラックリスト化、Windowsのファイルシステムのフィルタリングなどが重要になります。

6.4.1コンテナイメージベースのサーバーレスサービスにアクセスするためのAPIアクセスの管理

Kubernetesは、コンテナ化されたアプリケーションをデプロイするためのオープンソースのオーケストレーターです。Kubernetesは完全にAPI駆動型であるため、誰がクラスタにアクセスできるか、またどのようなアクションを実行できるかを制御・制限することが第一の防御策となります。Kubernetesクラスタは、シンプルな宣言型の構文でアプリケーションを定義してデプロイできるオーケストレーションAPIを提供します。Kubernetes APIは、ソフトウェアのゼロダウンタイム更新を容易にするデプロイ方法や、サービスのレプリカ数にトラフィックを容易に分散させるサービスロードバランサーなどの概念を公開しています。さらに、Kubernetesはサービスのネーミングとディスカバリのためのツールを提供しており、疎結合のマイクロサービスアーキテクチャを構築することができます。Kubernetesは、すべてのAPI通信がデフォルトでTLSで暗号化されることを期待しています。ノード、プロキシ、スケジューラ、ボリュームプラグインのようなインフラの一部であっても、すべてのAPIクライアントは認証されなければなりません。これらのクライアントは通常、サービスアカウントまたはx509クライアント証明書を使用しており、クラスタの起動時に自動的に作成されるか、クラスタのインストールの一部として設定されます。認証されると、すべてのAPIコールも認証チェックを通過することが期待されます。

Kubernetes Control plane componentsの説明は、このKubernetes.ioのウエブサイトにあります。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

6.4.2 コンテナイメージをベースにしたサーバーレスな設定とポリシーの実施

Kubernetesコントロールプレーン：

コントロールプレーンの主な構成要素には、Kubernetesを制御するためのREST APIを提供するKubernetes APIサーバーがあります。このAPIのフルパーミッションを持つユーザーは、クラスタ内のすべてのマシンで同等のルートアクセスが可能です。このAPIのコマンドライン/クライアントであるkubectlは、リソースやワークロードを管理するためにAPIサーバーへのリクエストを行うために使用できます。このKubernetes APIへの書き込みアクセス権を持つユーザーは、ルートユーザーとしてクラスタを制御することができます。APIサーバーがリクエストをリッスンするポートはすべて閉じて、ユーザーの適切な認証と認可を行う必要があります。詳細は「6.3.5 アクセス管理制御」を参照してください。Kubernetesのロールベースのアクセスコントロールは、Kubernetesリソース周辺のパーミッションを管理するための柔軟な認可ポリシーの設定に使用されます。

Kubernetes APIのエンドポイントは、インターネット上に公開してはいけません。エンドポイントは、Kubernetesクラスタがデプロイされているネットワーク内でのみアクセス可能であるべきです。Kubernetes APIサーバーは、リクエスト数に応じて拡張するように設定し、クラスタがトラフィックの大規模な急増に対応できるようにする必要があります。

kubeletは、各クラスタノード上のエージェントで、コンテナランタイムとやり取りを行うポッドを起動したり、ノードやポッドのステータスやメトリクスを報告したりします。クラスタ内のキューブレットにはAPIがあり、他のコンポーネントが統合され、ポッドの起動や停止などの命令を提供します。権限のないユーザーが任意のノードでこのAPIにアクセスし、クラスタ上でコードを実行することも想定されます。その場合、クラスタ全体をコントロールすることが可能になります。Kubelet APIへのアクセスを制限し、APIの--admission control 設定[1]でNodeRestrictionを実施することで、kubeletsのパーミッションを制限することが重要です。これにより、Kubeletが自分に関連付けられたポッドのみを変更できるように制限することができます。また、Kubeletsは、APIサーバーと通信するためにクライアント証明書を必要とします。これらの証明書は定期的にローテーションする必要があり、自動的にローテーションするように設定することもできます。

Kubernetesは、設定や状態の情報をetcdと呼ばれる分散型のキーバリューストアに保存します。etcdに書き込むことができるユーザーは、Kubernetesのクラスタを効果的にコントロールすることができます。etcdの内容を読むだけでも、攻撃者に役立つヒントを簡単に提供することができます。すべてのKubernetesのシークレットは、ETCDに保存される前に暗号化されるべきです。シークレットは、セキュリティのための保存時のETCDの暗号化に頼るべきではありません。Kubernetes APIは、高可用性とフォールトトレラントな配置として展開されるべきです。可用性の高いデプロイは、情報セキュリティの可用性要件を満たしており、停電や攻撃、インシデントが発生してもクラスタを管理でき、サービスの品質を維持できることを保証します。ETCD ノードは、ポート 2379 の API サーバーからのトラフィックのみを受け入れるように、セキュリティを設定する必要があります。また、ETCDノードは、他のETCDノードからの2380番ポートからのトラフィックのみを受け入れるように設定する必要があります。ETCDノードは、可用性の高いロードバランサの後ろに配置する必要があります。これにより、トラフィックが各ノード間で均等に分散され、ETCDノードがAPIサーバーに直接さらされないようになります。ETCDノードは奇数個配置し、クラスタ内のリーダーを適切に選出できるようにします。これは高可用性のために必要です。ETCDノードは、APIサーバーや他のKubernetesコンポーネントとは別のインスタンスにデプロイする必要があります。

[1] http://bit.ly/2IgwP7G

© Copyright 2021, Cloud Security Alliance. All rights reserved.

KubernetesのAudit、Authenticator、API、Controller Manager、Schedulerのログを保存し、問題がないか監視する必要があります。よくある問題は、繰り返される認証の失敗です。静的なパスワードを使用した基本的な認証は無効にする必要があります。Kubernetesクラスタは、企業のアイデンティティおよびアクセス管理と統合されている必要があります。KubernetesホストはパブリックIPアドレスを持つべきではありません。Kubernetesホストは、承認されたオペレーティングシステム上で実行されるべきです。[Kubernetes, 2021]を参照してください。

データプレーン

ポッドのセキュリティポリシーは、クラスタレベルで定義され、クラスタ内のすべてのポッドで使用される必要があります。デフォルトでは、ポッドは特権ユーザーとしての実行を許可されるべきではなく、パーミッションを権限昇格すべきではありません。KubernetesワーカーノードにはパブリックIPアドレスを割り当てるべきではなく、ロードバランサまたはプロキシサーバーの後ろでのみトラフィックを提供する必要があります。ポッドは、役割に基づいて少なくとも特権クレデンシャルを動的にプロビジョニングされます。これらのクレデンシャルは、バックエンドプロバイダと統合されたKubernetesのWebhookによって生成されるべきです。ポッドは、基盤となるホストの認証情報を継承してはいけません。サービスアカウントによって明示的に割り当てられていない限り、ポッドは基盤となるクラウドプロバイダに対してデフォルトで特権を持たないようにします。

すべてのポッドには、デフォルトで最も権限の弱いサービスアカウントまたはデフォルトのサービスアカウントを割り当てる必要があります。

ポッドとポッドの一部として実行されるコンテナを保護することは、コンテナ環境を保護するための重要な側面です。これらは、Kubernetesクラスタを攻撃するために使用される可能性のある個々のコンピュートユニットです。階層化防御のためには、最も低いコンポーネント単位のレベルでセキュリティを適用することで、個々のアプリケーションコンポーネントに実装されているよりきめ細かい制御が保証されます。
Kubernetesをはじめとするコンテナオーケストレーションシステムでは、ユーザーがポッドを強化・保護するための機能が提供されています。例えば、KubernetesのセキュリティコンテキストやPod Security Policiesなどのセキュリティポリシーがあります。

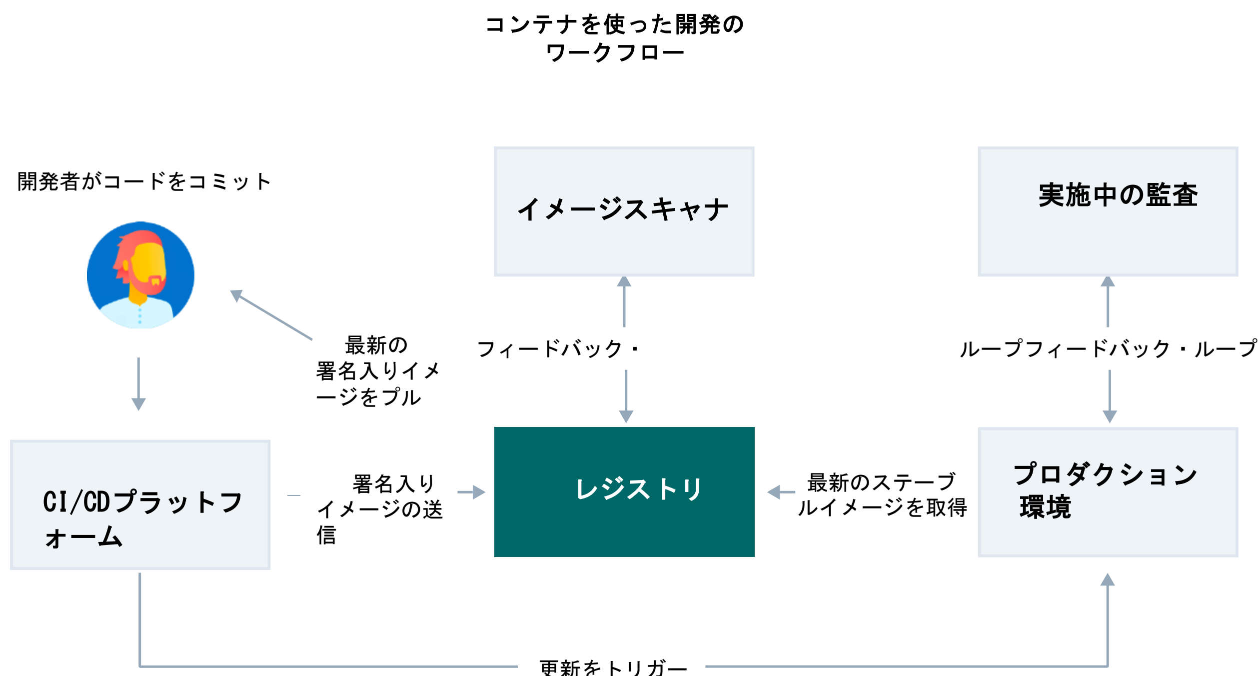
オープンソースのツールには、Open Policy Agent (OPA)などがあり、セキュリティポリシーを実施することができます。

これらのポリシーからの逸脱が必要な特定の要件があるかもしれません。このような場合、個々のコンテナのセキュリティコンテキストを指定できるように、所定のポッド内のすべてのコンテナにセキュリティポリシーを適用することが不可欠です。したがって、セキュリティコンテキストフィールドは、コンテナmanifestに含まれていなければなりません。個々のコンテナの制約は、重複や衝突がある場合、ポッドに指定された制約よりも優先されます。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

6.4.3 ベースイメージの管理とセキュリティ強化

ベースイメージ



ベースイメージは、すべてのイメージの基盤となるもので、ベースレベルのオペレーティングシステムに相当するものと考えられます。ベースイメージは、クラウドの開発チームや、組織内のOSチームが所有することができます。これらのイメージは、さまざまな公式リポジトリ（Docker、Quayなど）から取得され、組織のイメージレジストリ／リポジトリに保存されます。これらのイメージは、組織内で使用するために承認・認可されたバージョンとしてエンジニアリングチームが利用できるようにする必要があります。NIST 800-190によると、組織は可能な限り他のサービスや機能を無効にして、最小のコンテナ専用ホストOSを使用する必要があります。組織は、業界のベストプラクティスに基づいて、これらのイメージを（パッチ適用とハードニングによって）セキュリティ強化し、ハードニングベンチマーク（Centre for Internet Security - CIS Benchmarks）に基づいて、ビジネスニーズを満たすために必要なポート、プロトコル、およびサービスのみが提供されるようにする必要があります。

CI/CDパイプラインを使用して、ベースイメージの構築、タグ付け、テストを自動的に行います。さらに、開発チーム、テストチーム、セキュリティチームがイメージを本番環境にデプロイする前に署名することを義務付けることで、これを補完します。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

数少ないベースイメージセキュリティ・ハードニングのベストプラクティス

デフォルトのアカウントを削除または無効化	デフォルトのアカウント名とパスワードは、攻撃者のコミュニティでよく知られています。
OSユーザー認証の設定	ベースイメージは、想定されるユーザー／サービスを認証するように設定し、暗号化して使用します。可能な限りMFAを使用すべきです。組織は、信頼されていないネットワークを介して送信する際に、機密性の高い認証情報を保護するためにTLSを実装すべきです。
アクセス権の制限	RBACおよびABACアクセス管理を設定し、すべてのユーザーが自分のタスクを遂行するために必要な権限を持つことを保証します（それ以下でもそれ以上でもありません）。
否認と完全性	コンテナイメージ（イメージマニフェストを介して）は、notaryに安全に保管されているデジタル署名を使用して変更から保護する必要があります。
イメージ登録の設定	ベースイメージのアップストリームバージョンは、セキュリティの強化を行った後、組織内で使用するために承認・認可されたバージョンとしてテナントのエンジニアリングチームに提供されなければなりません。
イメージの脆弱性への対応	新しいセキュリティ脆弱性が発見された場合には、（組織のビジネス／コンプライアンス上の必要性に基づいて）パッチやその他の緩和策をできるだけ早く導入し、リスクが解決されたことを確認するために、脆弱性に対するテストをすぐに行う必要があります。
ソフトウェアサプライチェーンの確保	デプロイメントパイプラインの重要な側面は、デプロイメントが組織の承認されたプロセスに従っていることを確実にすることです。これを確立するためのいくつかの方法は、Binary Authorization、タグ、認証（attestation）などのものです。 コンテナイメージに対応するコミットハッシュをタグ付けすることで、イメージを特定の時点までさかのぼることが容易になります。 認証者（Attestor）は、デプロイメントパイプラインの重要な各段階においてコンテナレジストリにイメージに関するメモを追加することで、イメージが特定の基準を満たしているかどうかを表明することができます。 Binary Authorizationでは、クラスタ間でイメージをデプロイする前に、イメージに対する特定の認証を必要とするポリシーを設定することができます。定義されたポリシーが満たされない場合、イメージはデプロイされません。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

6.4.4 Kubernetesの設定とサービスメッシュポリシーの実施

Kubernetes（K8s）は、コンテナ化されたアプリケーションのデプロイ、スケーリング、管理を自動化するオープンソースのオーケストレーションプラットフォームです。Kubernetesコントロールプレーンは、クラスタ全体の状態と設定データを、永続的な分散型のキーバリューデータストアであるectdに格納します。各ノードはectdにアクセスすることができ、ノードはectdを通じて、実行中のコンテナの構成を維持する方法を知ります。

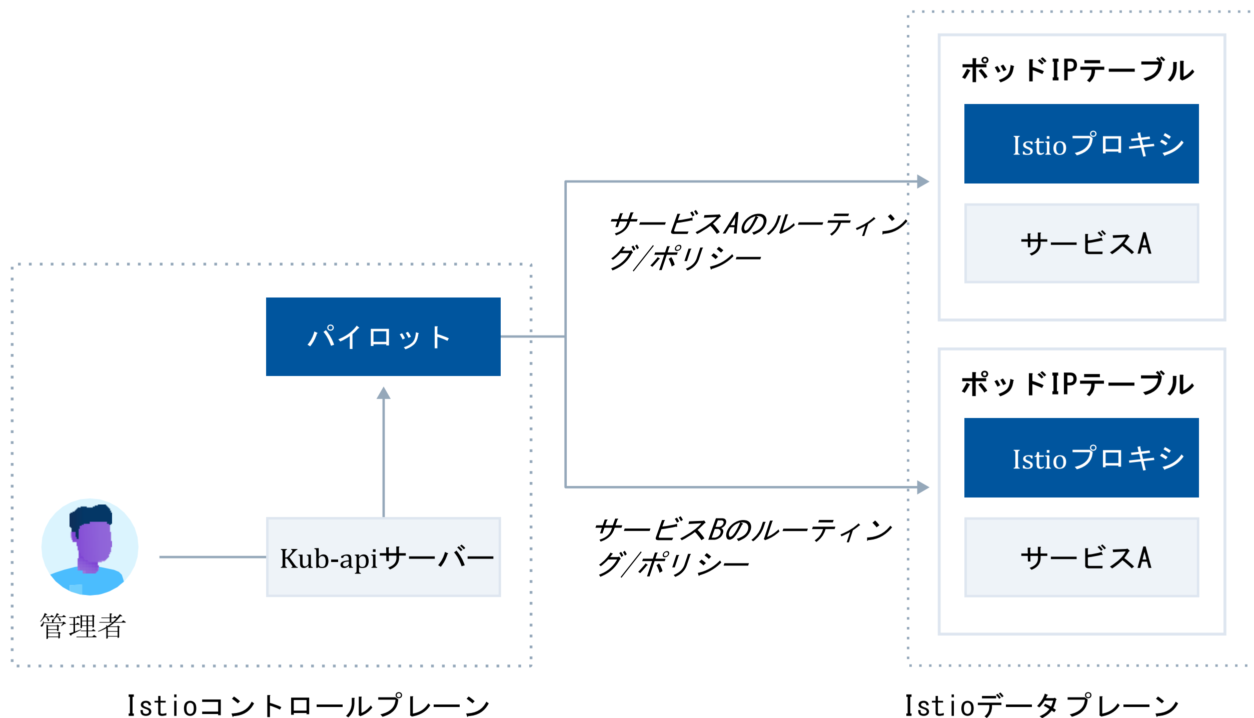
K8sのコントロールプレーンは、kube-Apiserverを介してクラスタ内のコンポーネントと通信します。etcdの設定が、クラスタ内で動作するコンテナの設定と一致するようにします。これは、宣言型APIによって行われます。しかし、K8sは設定言語/システム（JSON、YAML）の標準（または必須）を規定していません。

Kubernetesコントロールプレーンの中核となるのが、APIサーバーです。APIサーバーはHTTP APIを公開し、エンドユーザー、クラスタの他の部分、外部コンポーネントが相互に通信できるようにします。Kubernetes APIでは、Kubernetes内のAPIオブジェクト（Pod、Namespaces、ConfigMaps、Eventなど）の状態を問い合わせたり操作したりすることができます。ほとんどの操作は、RESTコール経由でAPIを使用するkubectlコマンドラインインターフェースで実行できます。Network Policyは、Kubernetesの機能の1つで、ポッドのグループがどのようにお互いや他のネットワークエンドポイントとの通信を許可されるかを設定するものです。ネットワークポリシーは、OSIモデルのレイヤー3（ネットワーク）とレイヤー4（トランスポート）で動作します。Kubernetesのネットワークポリシーは、Kubernetesのワークロードのグループ（以下、ポッドと呼ぶ）が、相互および他のネットワークエンドポイントとの通信を許可される方法を指定します。ネットワークポリシーのリソースは、ラベルを使用してポッドを選択し、選択されたポッドに対してどのようなトラフィックを許可するかを指定するルールを定義します。

ポッドのIPアドレスは永続性がなく、スケールアップやスケールダウン、アプリケーションのクラッシュ、ノードの再起動などに応じて現れたり消えたりします。この問題を解決するために、Kubernetesにはサービスが組み込まれています。Kubernetesのサービスは、一連のポッドの状態を管理し、時間とともにダイナミックに変化するポッドの一連のIPアドレスを追跡することができます。サービスはPodを抽象化したものとして機能し、PodのIPアドレス群に単一の仮想IPアドレスを割り当てます。サービスの仮想IPに割り当てられたトラフィックは、その仮想IPに関連付けられている一連のPodにルーティングされます。これにより、サービスに関連付けられた一連のPodをいつでも変更することができます。クライアントは、変更されないサービスの仮想IPだけを知っていればよいです。

Istioのサービスメッシュは、サポートされているプロトコルに対してポリシーベースのルーティングとポリシーベースの承認を提供するサービスレイヤーとしてのセキュリティです。Istioのポリシーは、「サービス」（OSI Layer 7）で動作します。サービスメッシュは、プロキシとノードエージェントのコンテナをアプリケーションのデプロイ仕様に入れ、ポッドのネットワーク名前空間を共有し、透過的にトラフィックを傍受してポリシーを適用します。ノードエージェントは、ポッドのサービスアカウントトークンを使用してプロキシのプラットフォーム証明書を取得し、プロキシのシークレットディスカバリーサービス（SDS）を使用して証明書/鍵をプロキシに提供します。サービスメッシュに対応したポッドの場合、ポッドに出入りするすべてのTCPトラフィックは、ポッド内のサイドカーコンテナとして動作する envoy プロキシを介してリダイレクトされ、下図のように外部サービスをサポートするように拡張することができます。

© Copyright 2021, Cloud Security Alliance. All rights reserved.



Istioを使ったKubernetesのポリシーの実施

Istioの中のサービスとサービスの間：
どちらのサービスも単一のクラスタのサービスメッシュ内にあり、Kubernetesのネイティブサービスとしてデプロイされているため、istioコントロールプレーンが自動的に定義を発見し、プロキシを適切に設定することができます。

外部サービスからIstioへ：
これは、メッシュ外のクライアント（curlなど）やサービスが、メッシュ内のサービスに接続する様子を表しています（上の図ではSvcA-SvcBと表示されています）。この通信が機能するためには、クライアントはクライアント証明書を必要とし、SvcBもIstioのIngress gatewayと統合されている必要があります。

Istioから外部サービスへ：
この場合、メッシュ内のクライアントは、図中でSvcC-to-SvcDと示されている外部サービスへの接続を開始する必要があります。

Istio Proxy/Envoyサイドカーに加えて、Githubブログで提案されているように、アプリケーションポッドにOPAサイドカーを含めることができます。Istio Proxyは、マイクロサービス宛のAPIリクエストを受信すると、OPAに確認してリクエストを許可すべきかどうかを判断します。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

6.4.5 アクセス管理コントロール

Apiserverは、セキュアなHTTPSポート（443）でリモート接続をリッスンするように設定されており、1つまたは複数の形式のクライアント認証が有効になっています。また、1つまたは複数の形式の認証を有効にする必要があります。

認証/認可

コントロールプレーン内のすべての通信は、相互に認証されたTLSを介して行われます。APIへの外部からのHTTPSアクセスを許可するファイアウォールルールが設定されています。ユーザーは、kubectl、クライアントライブラリ、またはRESTリクエストを使用してAPIにアクセスします。APIへのアクセスには、人間であるユーザーとKubernetesのサービスアカウントの両方が認証されます。リクエストがAPIサーバーに到着すると、APIサーバーは一連のチェックを行い、リクエストに対応するかどうか、また対応する場合は**定義されたポリシー**に基づいてさらに検証するかどうかを決定します。認証は通常、K8sのRBAC認証モジュールによって実装されます。しかし、Webhook認可モジュールを活用したOpenPolicyAgent(OPA)による高度な認可ポリシーなど、別の方法で実装することも可能です。

アドミッションコントローラ/ポッドセキュリティポリシー

APIリクエストが認証され認可されると、リソースオブジェクトは、アドミッションコントローラを使用して、クラスタの状態データベースに永続化される前に、検証または変異の対象とすることができます。アドミッションコントローラは、オブジェクトの永続化の前に、リクエストが認証・認可された後に、Kubernetes APIサーバーへのリクエストをインターセプトするコードの一部です。
いくつかのアドミッションコントローラを以下に紹介します：

- **DenyEscalatingExec** – ポッドが、強化された特権で実行できるようにする必要がある場合（たとえば、ホストの IPC/PID 名前空間を使用するなど）、このアドミッションコントローラは、ユーザーがポッドの特権コンテナでコマンドを実行するのを防ぎます。
- **PodSecurityPolicy** – 作成されたすべてのポッドにさまざまなセキュリティメカニズムを適用する手段を提供します。PodSecurityPolicyオブジェクトは、ポッドがシステムに受け入れられるために実行しなければならない一連の条件と、関連フィールドのデフォルトを定義します。
- **NodeRestriction** – kubeletのクラスタリソースへのアクセスを管理するアドミッションコントローラです。
- **ImagePolicyWebhook** – ポッドのコンテナに定義されたイメージを、外部の「イメージバリデータ」によって脆弱性をチェックできるようにします。

Open Policy Agent Gatekeeper（CRD）

Kubernetesでは、リソースが作成、更新、削除されるたびに実行されるアドミッションコントローラのWebhookによって、ポリシーの決定をAPIサーバーの内部から切り離すことができます。Gatekeeperは、ポリシーエンジンであるOpen Policy Agentが実行するCRDベースのポリシーを実施するための検証用Webhookです。また、Gatekeeperの監査機能により、管理者はどのリソースが現在ポリシーに違反しているかを確認することができます。さらに、Gatekeeperのエンジンは、infrastructure-as-codeシステムの信頼の源(source of truth)へのコンプライアンス違反のコミットを検出・拒否することができ、コンプライアンスへの取り組みをさらに強化します。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

Istioサービスメッシュ - 認証

Istioは、クライアントとサーバーのプロキシが証明書を交換するサービス間の相互TLS認証をサポートしています。証明書はプロキシによってCAに対して検証され、証明書からSPIFFE（Secure Production Identity Framework for Everyone）URIが抽出され、相手のIDとして使用されます。この標準的な認証メカニズムにより、CAと統合されたサービスメッシュの外部にあるクライアントやサーバーと相互運用することができます。相互TLSを有効にすることは、クラスタ、ネームスペース、およびサービスレベルで設定できます。認証ポリシーを使用して、クライアント証明書の認証を実施することができます。Istioは追加のクライアント認証方法もサポートしています（例：JWT）。

Istioサービスメッシュ - 認可

Istioの認可機能は、サービスメッシュで実現したサービスに対して、ネームスペースレベル、サービスレベル、メソッドレベルのアクセス制御を行います。特徴は以下の通りです：

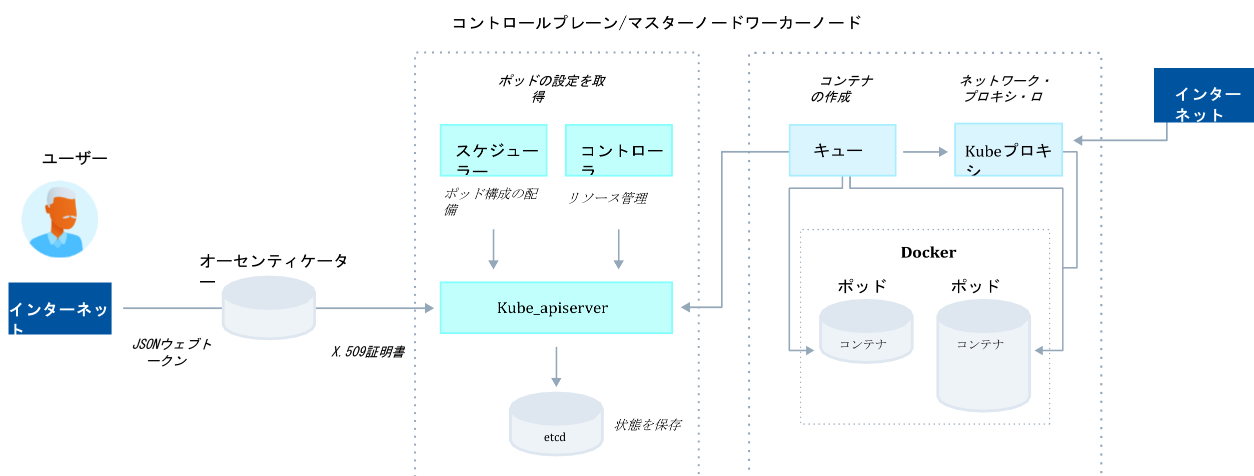
- シンプルで使いやすいロールベースのセマンティクス
- ローカルのenvoy proxyでネイティブに認可が行われるため、ハイパフォーマンスな運用が可能
- HTTPおよびGRPCプロトコル属性とプレーンTCP接続のサポート

ネームスペース管理者は、自分のネームスペース内のサービスに対する認可ポリシーを構成することができます。ポリシーは、k8s APIを使用してk8sカスタムリソース定義（CRD）として構成されます。他のk8s CRDと同様に、認可ポリシーはネームスペースで構成されています。つまり、サービス管理者がサービスの認可ポリシーに影響を与えるには、サービスのk8sネームスペースにあるポリシーCRDへの書き込みアクセスが必要です。

サービスに対して認可を有効にすると、ポリシー（*誰がどのような条件で何を*できるか）で許可されていない限り、サービスへのインバウンドリクエストはすべて失敗します。

6.4.6 Kubernetesリスクとコントロール

例：APIサーバーへのアクセス



© Copyright 2021, Cloud Security Alliance. All rights reserved.

APIサーバーは、クラスタ内のすべての通信のハブとなります。クラスタのセキュリティ設定の大部分が適用されるのもAPIサーバーです。仮に何者かがAPIに未承諾でアクセスしたとします。その場合、あらゆる種類の機微情報を取得したり、クラスタ自体を制御したりすることが可能になるかもしれません。APIサーバーへのアクセスを慎重に管理することは、セキュリティの観点から非常に重要です。

APIサーバーは、K8sクラスタ上のすべてのワークロードと、それらが使用するリソースへのアクセスをコントロールします。クラスタのセキュリティ設定の大部分が適用されるのは、APIサーバー上です。もし、悪者がK8s APIサーバー経由で特権リソースへの不正アクセスを得た場合、悪者はK8s RBACで保護されているあらゆるクラスタリソース（例：シークレット）へのアクセスや、クラスタノード上のあらゆるファイルへのアクセスなどが可能になります。したがって、特権リソースへのアクセスを慎重に管理することは、クラスタの完全性を維持するために非常に重要です。Kubernetesにデプロイされた新しいアプリケーションの最も基本的なニーズの1つは、いくつかのウェブエンドポイントを異なるURLで公のインターネット上に公開し、SSLで保護することです。そのためには、エンドポイントのセキュリティを管理することが重要です。

以下のリストは、APIサーバーへのアクセスが保護されていない場合に発生する可能性のあるエクスプロイトの例です：

- サービス拒否攻撃（DOS）とは、被害者のサービスが多くの偽のリクエストで過負荷状態になることです。その結果、サービスは正当な要求に応答することができなくなります。極端なケースでは、この攻撃によってマシン全体のシステムリソース（CPU、メモリ、ネットワーク帯域幅、ディスクI/Oなど）が枯渇し、さらに被害が拡大します。
- Netflixは、HTTP/2の実装に影響する8つの脆弱性をアドバイザリで発表しており、そのうち2つはGo net/httpライブラリに影響します。これは、CVE-2019-9512 "Ping Flood" および CVE-2019-9514 "Reset Flood" です。net/httpパッケージを使用してHTTP/2リクエストをリッスンするGoで書かれたアプリケーションは、Kubernetesを含めてDoS攻撃を受ける可能性があります。
- CVE-2019-9512 「Ping Flood」: 攻撃者は HTTP/2 リスナーを連続した ping リクエストの流れで攻撃します。受信者は – 各リクエストに応答するために – 次々と応答をキューに入れ始めます。このキューは成長し、アプリケーションがクラッシュするまで、より多くのメモリとCPUを割り当てます。
- CVE-2019-9514 "Reset Flood": この攻撃は、HTTP/2 の RST_STREAM フレームを利用している点を除いて、最初の攻撃と同様のテーマを持っています。RST_STREAM は、単純に、あるピアから送信されると、他のピアに接続をキャンセルする必要があることを知らせるフレームタイプです。そのため、サーバーに複数のストリームを開き、それらを通じて無効なデータを送信することで、DoS攻撃を仕掛けることができます。無効なデータを受信したサーバーは、「無効な」接続をキャンセルするために、RST_STREAMフレームを攻撃者に送信します。たくさんのRST_STREAMレスポンスがあると、それらはキューに入り始めます。キューが膨大になると、アプリケーションがクラッシュするまで、より多くのCPUとメモリがアプリケーションに割り当てられます。
- CVE-2018-1002105 – kube-apiserver のプロキシされたアップグレード要求に対するエラー応答の不適切な処理。
- CVE-2016-7054 (OpenSSLadvisory) [重大性が高い] – *-CHACHA20- POLY1305 暗号スイートを使用する TLS 接続は、より大きなペイロードを破損することで DoS 攻撃を受ける可能性があります。
- 「2018-2019 Global Application & Network Security」レポートによると、2018年に最も多く報告されたアプリケーション層への攻撃形態は、SSL/TLSを悪用したHTTPSフラッド攻撃またはDDoS攻撃でした。SSL/TLSを使用することで、私たちは真正性、完全性、機密性を享受することができます。しかし、送信先のサーバーに関して言えば、SSL/TLS接続は大量の割り当てられたリソースを必要とします。正確には、リクエストしたホストからの15倍のリソースを必要とします。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

例：設定ミス

Kubernetesは複雑なシステムであり、複数の設定オプションがあります。これらの設定は、セキュアに実施する必要があります。セキュリティの誤設定は、機密性の高いユーザーデータやシステムの詳細を公開し、サーバーの完全な侵害につながる可能性があります。セキュリティの誤設定は、セキュアでないデフォルトの設定、不完全またはアドホックな設定、オープンなクラウドストレージ、誤って設定されたHTTPヘッダー、不要なHTTPメソッドなどに起因します。攻撃者は、これらの設定ミスを利用して、パッチが適用されていない欠陥、共通のエンドポイント、または保護されていないファイルやディレクトリを悪用し、不正なアクセスを行います。

緩和策：

- 再現性のあるハードニングとパッチ適用のプロセスを確立する。
- 不要な機能を無効にする。
- 管理者権限でのアクセスを制限する。
- エラーを含むすべてのアウトプットを定義し、実施する。
- ホストがセキュアであり、正しく構成されていることを確認する。CISベンチマークに照らし合わせて設定を確認する。
- Autocheckerは、これらの規格への適合性を自動的に評価します。
- 適切にロックダウンされた環境を迅速かつ容易に導入するための反復可能なハードニングプロセスを確立すること。
- Kubernetesは、設定や状態の情報を「etcd」と呼ばれる分散型のキーバリューストアに保存する。etcdに書き込むことができるユーザーは、Kubernetesクラスタを効果的にコントロールすることができる。
- APIスタック全体の構成を見直し、更新する。このレビューには、オーケストレーションファイル、APIコンポーネント、クラウドサービスが含まれる。
- 自動化されたプロセスにより、すべての環境における設定の有効性（設定の欠陥）を継続的に評価する。
- カスタムダッシュボードやアラートを設定することで、疑わしい活動を早期に検知して対応することを可能にする。
- インフラ、ネットワーク、APIの機能を継続的に監視する（場合によっては自動化による）。

例：脆弱性スキャン

イメージのライフサイクルを通じて脆弱性をスキャンすることは非常に重要です。また、スピードを維持することと組織のリスク許容度を比較検討することも重要です。組織は、イメージのセキュリティと脆弱性管理を取り扱うための独自のポリシーと手順を作成する必要があります。脆弱性のあるコンテナイメージをデプロイすると、その脆弱性を利用しようとする脅威ベクターによる攻撃が行われる可能性があります。

ベストプラクティス：

まず、安全でないイメージを構成する基準を、以下のような指標を用いて定義することから始めます：

- 脆弱性の深刻度
- 脆弱性の数
- それらの脆弱性にパッチや修正プログラムがあるかどうか
- 脆弱性が設定ミスを持ったデプロイに影響するかどうか

© Copyright 2021, Cloud Security Alliance. All rights reserved.

6.4.7 追加のセキュリティ

6.4.7.1　Kubernetesのセキュリティ・ベストプラクティス

Kubernetesは、コンテナ化されたアプリケーションのデプロイ、スケーリング、および管理を自動化するためのオープンソースのコンテナオーケストレーションエンジンです。KubernetesのK8sクラスタは、アプリケーションをホストするワーカーノード/ポッドで構成されています。Kubernetesのコントロールプレーンは、クラスタ内のポッドネットワークを管理します。ここでは、Kubernetesとコンテナのセキュリティに関するベストプラクティスを以下の4つのセクションに分けて説明します：

パート1

TLS everywhere
TLSをサポートするすべてのコンポーネントでTLSを有効にして、トラフィックの盗聴を防ぎ、接続の両側のアイデンティティを認証する必要があります。

サービスメッシュの実行
サービスメッシュとは、高性能な〝サイドカー〝プロキシサーバー「Envoy」と、そのプロキシサーバー間で行われる暗号化された持続的な接続のウェブのことです。これにより、マイクロサービスを変更することなく、トラフィック管理、モニタリング、ポリシーを追加することができます。

ネットワークポリシーの使用
デフォルトでは、Kubernetesのネットワークはすべてのポッド間のトラフィックを許可していますが、Network Policyを使って制限することができます。

Open Policy Agent（OPA）の使用
OPAを使用すると、Kubernetes APIサーバーを再コンパイルまたは再構成することなく、Kubernetesオブジェクトにカスタムポリシーを適用することができます。

ロギング＆モニタリング
アプリケーションやインフラレベルでの異常を検知するために必要です。攻撃や異常（高い使用率や潜在的な危険性）があれば、ロギングとモニタリングによって検出されます。
アプリケーションパフォーマンスモニタリングは、DDOS攻撃などの他の侵害を検出します。

Bastionホストの使用を検討
これは、攻撃に耐えられるように特別に設計、構成されたネットワーク上の特別な目的のコンピュータです。マスターへのアクセスは、Kubernetesへの特権的なアクセスを行うユーザーを監視することができるハード化されたBastionホストを介してのみ行われるべきです。

プライベートネットワーク
Kubernetesのマスターノードとワーカーノードは、企業ネットワークとのセキュアな接続を確保し、インターネットからの直接アクセスを防ぎ、全体的な攻撃対象を減らすために、プライベートサブネット上にデプロイする必要があります。

Linuxのセキュリティ機能の利用
Linux カーネルにはいくつかのセキュリティ拡張機能（SELinux）があり、アプリケーションに最低限の権限を与えるように構成することができます。Linux カーネルには、アプリケーションに与える特権が少なくなるように設定できる、多くの重複するセキュリティ拡張機能（capabilities、SELinux、AppArmor、seccomp-bpf）があります。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

クラスタノードイメージ
Kubernetesクラスタで構築する場合、Linuxのイメージを使用することになります。それをCISベンチマーク対応にして、Linuxのセキュリティ管理がすべて行われていることを確認する必要があります。OSのハードニングを行わないと、インフラにソフトウェアの脆弱性が発生する可能性があります。また、ソフトウェアのサプライチェーン全体のセキュリティを考慮し、セキュリティのベストプラクティスに沿ってソフトウェアの管理の連鎖（chain of custody）が行われていることを確認することも重要です。Binary Authorization、タグ、証明書などは、その実装方法の一例です。

etcdクラスタの分離とファイアウォール
etcdは状態とシークレットの情報を保存し、Kubernetesの重要なコンポーネントであるため、クラスタの他の部分とは異なる方法で保護する必要があります。

暗号化鍵のローテーション
セキュリティのベストプラクティスは、暗号化鍵と証明書を定期的にローテーションすることで、鍵が漏洩した際の「影響範囲」を限定することです。

パート2：コンテナセキュリティ

PodSecurityPolicyの使用
ポリシーは、セキュリティの重要な要素ですが、見落とされがちな部分であり、同時に検証と変異の対象のコントローラとして機能します。PodSecurityPolicyは、APIを通じてデフォルトで有効化されるオプションのアドミッションコントローラです。したがって、ポリシーは、PSPアドミッションプラグインが有効でなくても展開できます。

YAMLの静的な解析
PodSecurityPolicyがAPIサーバーへのアクセスを拒否する場合、組織のコンプライアンス要件やリスク許容度をモデル化するために、開発ワークフローで静的解析を使用する必要があります。

非root ユーザーとしてコンテナを実行
ルートで実行されるコンテナは、ワークロードが必要とする権限よりもはるかに多くの権限を持っていることが多く、これが侵害された場合には、攻撃者がさらに攻撃を進めることになりかねません。

パート3：セキュリティの自動化

イメージスキャン
すべてのデプロイメントは、自動化されたCI/CD環境によって制御されなければなりません。大まかにいうと、Kubernetesではすべてがコンテナイメージとしてデプロイされます。誰かがアプリケーションを作成すると、それはコンテナイメージとなり、コンテナレジストリに追加されます。コンテナイメージのスキャナは、CI/CDパイプラインの一部である必要があります。誰かがコンテナイメージを作成したら、そのコンテナイメージに対して継続的に脆弱性のスキャンを行う必要があります。イメージのホワイトリスト化は、Kubernetesのアドミッションコントローラで行うことができます。アプリケーションが特定のイメージを使用する場合、それらを承認する必要があります。

シークレット管理
また、クラスタはシークレット管理システムを介して統合される必要があります。これにより、アプリケーションポッドは、ポッドに装備されているAppRoleに基づいて、実行時に必要なパスワードとシークレットを自動的に受け取ることができます。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

コード解析
コードスキャンと静的コード解析もセキュリティの自動化には欠かせない要素です。Kubernetesで何らかのアプリケーションコードを扱う際には、ソースコードをスキャンして、脆弱性やハードコードの異常がないことを確認する必要があります。

第三者による脆弱性スキャン
脆弱性の評価は、組織にとって中核的な要件です。これは、脆弱性を持つ可能性のあるアップストリームコンポーネントを使用している場合に特に必要です。サードパーティ製の脆弱性スキャン(BlackDuck、Tenableなど)を継続的に実施することで、組織は脅威となるエージェントの一歩先を行くことができます。これらの標準的な脆弱性スキャンツールは、オープンソースのライブラリをスキャンして、さまざまな脆弱性データベース（National Vulnerability Databaseなど）に掲載されている既知の脆弱性を検出します。

脆弱性が特定されると、それらが組織の業務、ビジネス、規制要件、または評判にどのような影響を与えるかによって、その修復方法が異なります。例：PCI DSS 3.2.1 では、高/重大タイプの既知の脆弱性を30 日以内に修正することが求められています。

DevSecOps（CI/CD）
セキュリティは、DevSecOpsのプロセス全体に組み込まれている必要があります。また、DevSecOpsを支えるアジャイルプロセスもセキュアでなければならず、security user storiesがバックログに入っていなければなりません。ソフトウェアのライフサイクル全体にセキュリティを組み込むことで、脆弱性を早期に発見し、迅速な修正を行い、その結果、コストを削減することができます。
デバイス、ツール、アカウントなどの継続的なモニタリングは、継続的な発見・検証につながります。

パート4：アイデンティティ／アクセス管理

最小権限でRBACを有効化
ロールベースアクセスコントロール（RBAC）は、ネームスペースへのアクセスなど、リソースへのユーザーのアクセスに対してきめ細かなポリシー管理を行います。組織全体で認証と認可を一元化する（シングルサインオン）ことで、ユーザーのオンボーディング、オフボーディング、一貫したパーミッションを実現します。

APIサーバーにサードパーティの認証を使用
Kubernetes - アクセスの制御。Kubernetes APIへのアクセスを安全に管理するために、管理者は詳細な認証・認可ポリシーを作成し、高度でフル機能のスクリーニング技術を実装する必要があります。

6.4.7.2 APIセキュリティ（OWASPトップ10）

大規模なコンテナ管理を自動化するには、APIが不可欠です。APIは以下のように使用されます：

- ポッド、サービス、レプリケーション・コントローラのデータを検証し、設定する。
- 受信したリクエストの検証を行い、他のコンポーネントのトリガーを呼び出す。

APIの脅威は、最も一般的な攻撃手段と考えられています。脅威となるエージェントは、API に存在するこれらの脆弱性を利用して、アプリケーションに侵入します。OWASPTop10threatsandmitigationsを参照してください。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

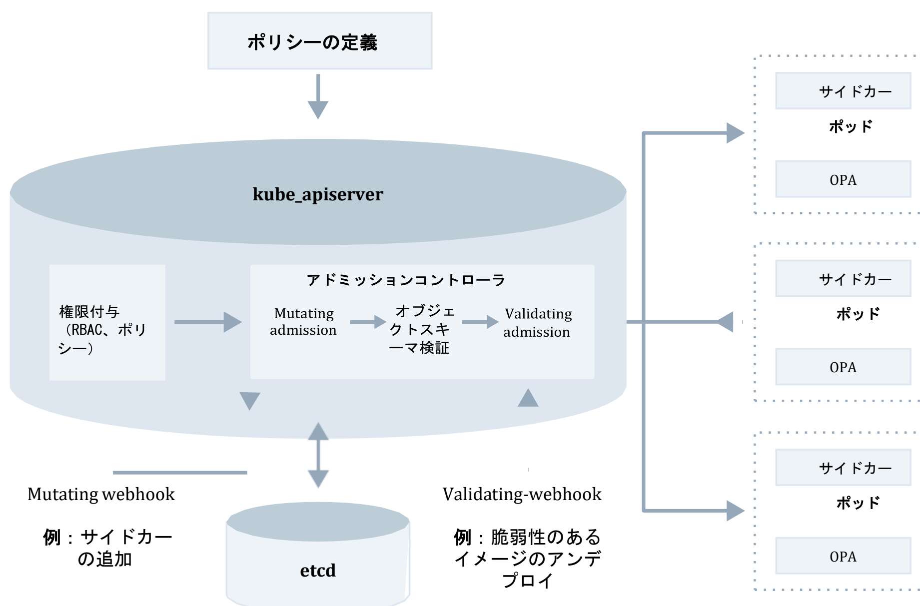
6.4.7.3 Kubernetesポリシー

Kubernetesでは、AdmissionControllerWebhooksによって、ポリシー決定をAPI Serverの内部動作から切り離すことができます。Admission Webhookは、アドミッションリクエストを受け取り、それに対して何かを行うHTTPコールバックです。Admission Webhookには、validation admission webhookとmutating admission webhookの2種類があります。Mutating admission webhookは最初に起動され、APIサーバーに送信されたオブジェクトを変更して、カスタムのデフォルトを強制することができます。すべてのオブジェクトの修正が完了し、受信したオブジェクトがAPIサーバーによって検証された後に、validating admission webhookが呼び出され、カスタムポリシーを実施するためにリクエストを拒否することができます。

Open Policy Agent（OPA）は、オープンソースの汎用ポリシーエンジンで、スタック全体でのポリシー実施を一元化します。ハイレベルな宣言型言語を提供しており、コードとしてポリシーを指定したり、シンプルなAPIを使用してソフトウェアからポリシーの意思決定を切り離したりすることができます。OPAは、マイクロサービス、Kubernetes、CI/CDパイプライン、APIゲートウェイなどでポリシーを実施することができます。Gatekeeperは、OpenPolicyAgentで実行されたCRDベースのポリシーを実施するための検証（Mutating）用Webhookです。Gatekeeperの監査機能により、管理者はどのリソースが現在ポリシーに違反しているかを確認することができます。

Webhookを検証する例（LINK）

- 特権のあるコンテナの実行を禁止する
- ホストの名前空間の共有を禁止する
- ホストのネットワークやポートの使用をすべて制限する
- ホストファイルシステムの使用を制限する
- Linuxの機能をデフォルトのセットに制限する
- 定義されたボリュームタイプの使用を制限する
- rootへの権限昇格
- コンテナのユーザーIDとグループIDを制限する
- ポッドのボリュームを所有するFSグループの割り当てを制限する
- seccompプロファイルが必要



Kubernetesポリシーのベストプラクティス。

- すべてのネットワーク接続は、ポリシーに基づいて実施されるべきです。
- リモートエンドポイントのアイデンティティを確立するには、常に強力な暗号によるアイデンティティの証明を含む複数の基準に基づいています。
- IPアドレスやポートなどのネットワークレベルの識別子は、敵対するネットワークによって偽装される可能性があるため、それだけでは十分ではありません。
- 侵害されたワークロードが、ポリシーの適用を回避できないようにする必要があります。
- 暗号化して、ネットワークトラフィックを盗聴する者にデータが漏洩しないようにします。
- まず、「default-deny-all」ネットワークポリシーを適用してください。他のネットワークポリシーで明示的にホワイトリストに登録されている接続のみを許可します。
- ネットワークポリシーはネームスペースで構成されており、ネームスペースごとに作成されます。
- 外部からのトラフィックを受信するために、インターネットからのアクセスを許可するためにポッドに適用するラベルを指定し、そのラベルを対象としたネットワークポリシーを作成します。
- より厳密なポリシーを設定したい場合は、より細かいCIDRブロックを指定したり、許可されるポートやプロトコルを明示的にリストアップしたりすることが理想的です。

6.5 コンプライアンスとガバナンス

クラウドコンピューティングモデルを活用すると、企業はハードウェア、施設、ユーティリティ、データセンタへの投資を削減できるだけでなく、理論的にはこのリスクをCSPに移すことで全体的なリスクを削減できるはずです。クラウドプロバイダは、プラットフォームセキュリティに継続的に投資し、これらのリスク範囲を管理しているという大まかな前提がありますが、それをどうやって確認するのでしょうか。クラウドプロバイダのパフォーマンスとサービスの質を継続的かつ定期的に評価することは、組織のセキュリティ保証プログラムにとって不可欠な要素です。組織のソーシングチームは、クラウドプロバイダの状態を定期的に分析・評価し、契約上の義務が果たされていることを確認する必要があります。

セキュリティチームは、開発チームがデプロイしているサーバーレスアプリケーションに関連するリスクテーマを特定することに注力する必要があります。サーバーレスコンピューティングに関連するリスクやリスクテーマを分析することで、全体的なリスクについての理解を深めることができます。そして、リスクステートメントを作成し、特定のリスクから生じる可能性の高い被害を判断し、そのリスクに対して何をすべきかの評価を開始することができます。残存するリスクは、組織のGRCシステムに記録することができます。

すべての資産を把握することなく、環境を保護・管理することは困難です。

6.5.1サーバーレスのためのアセットマネジメント

アセットマネジメントシステムには、クラウドコンピュートの一過性の性質に対応した仕組みが必要です。クラウドコンピュートの一時的な性質は、資産管理とインシデント対応における課題を生み出します。
最良のアプローチの1つは、自動化されたアセットトラッキングをCI/CDパイプライン内に統合することです。こうすることで、新しいサーバーレス機能がデプロイされたり変更されたりするたびに、対応する変更が記録され、更新され、それに応じてアセットにタグが付けられます。モニタリング、アラート、監査のアクションや環境への変更がリアルタイムで更新されます。そして、この情報をシステムに統合することで、自動的に対応し、行動を起こすことができます。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

このようにして、可視化されたギャップを解消し、実際にインシデントが発生した場合の混乱や誤った対応を回避することができます。また、資産のインベントリを収集する代替的な方法として、ポーリング方式を採用することもできます。サーバーレスでは、組織のために有効化されたAPIが1日に一定回数ポーリングされます。ポーリングで得られた情報は、連続する2回のポーリング実行の差分を取ることで、組織の資産インベントリに更新されます。

クラウドコンピューティングモデルを活用すると、組織はハードウェア、施設、ユーティリティ、データセンタへの投資を削減することができ、理論的にはこのリスクをCSPに移すことで全体的なリスクを削減することができます。クラウドプロバイダは、プラットフォームセキュリティに継続的に投資し、これらのリスク範囲を管理しているという大まかな前提がありますが、それをどうやって確認するのでしょうか。クラウドプロバイダのパフォーマンスとサービスの質を継続的かつ定期的に評価することは、組織のセキュリティ保証プログラムにとって不可欠な要素です。組織のソーシングチームは、クラウドプロバイダの状態を定期的に分析・評価し、契約上の義務が果たされていることを確認する必要があります。

セキュリティチームは、開発チームがデプロイしたサーバーレスアプリケーションに関連するリスクテーマを特定することに注力しなければなりません。サーバーレスコンピューティングに関連するリスクやリスクテーマを分析することで、全体的なリスクの理解を深めることができます。そして、特定のリスクから生じる可能性の高い損害を決定し、そのリスクに対して何をすべきかの評価を開始するために、リスクステートメントを形成することができます。残存するリスクは、組織のGRCシステムに記録することができます。

サーバーレスコンピューティングの性質上、開発者はもはやインフラ、ネットワーク、ホストのセキュリティを気にする必要はありません。しかし、新たな攻撃手段が出現しています。その全容については、Cloud Security Allianceの「Top 12 Risks for Serverless Applications」を参照してください。

6.5.2サーバーレスガバナンス

ガバナンスの観点から、ファンクション/コンテナのインベントリはこのドキュメントのどこかで言及されるべきです。なぜなら、これはユーザーが過去に行う必要がなかったユニークな追跡であり、新しいツールや考え方が必要だからです。

ガバナンスの要素：

- 資産のインベントリ – CI/CDパイプライン、CMDB、CSPが管理する資産のメタデータ履歴を組み合わせて活用し、保護が必要な資産の全体像を把握します。
- 責任の共有 – お客様またはアプリケーションの所有者とCSP – ステークホルダーと規制当局の義務
- RACI – Responsibility, accountability, Consulted, Informed
- メトリクス – 測定方法
- オートメーション
- パフォーマンス – 効率と効果
- サービス合意書 – SLA – TOR（訳注： TORはTerms of Reference（参照条件）と思われる）

© Copyright 2021, Cloud Security Alliance. All rights reserved.

1. ステートレスな単一目的関数の開発
関数はステートレスであり、限られた期間のみ存続するものであるため、関数には単一目的のコードを書くことが推奨されます。これにより、関数の実行時間が制限され、コストに直接影響します。さらに、単一目的のコードは、テスト、デプロイ、リリースが容易であるため、企業のアジリティが向上します。最後に、ステートレスは制限があると思われるかもしれませんが、（ステートレスは）従来考えられなかったリクエストの増加に対応するために、プラットフォームに無限のスケーラビリティを提供します。

2. プッシュベース、イベントドリブンのパターンを設計
プッシュベースやイベントドリブンのアーキテクチャパターンを設計することで、ユーザーが入力しなくてもイベントの連鎖が伝播し、アーキテクチャにスケーラビリティをもたらします。

3. テクノロジースタック全体に適切なセキュリティメカニズムを組み込む
適切なセキュリティメカニズムは、APIゲートウェイのレイヤーだけでなく、FaaSのレイヤーにも組み込む必要があります。これらのセキュリティメカニズムには、アクセスコントロール、認証、アイデンティティ／アクセス管理、暗号化、信頼関係の確立などが含まれます。

4. パフォーマンスのボトルネックを特定
パフォーマンスのボトルネックを継続的に測定し、どの機能が特定のサービスの速度を低下させているかを特定することは、最適なカスタマーエクスペリエンスを確保するために重要です。

5. 厚みのあるパワフルなフロントエンドを作成
より複雑な機能をフロントエンドで実行する場合、特にリッチなクライアントサイドアプリケーションフレームワークを使用することで、関数の呼び出しや実行時間を最小限に抑え、コストを削減することができます。バックエンドのロジックをフロントエンドから完全に切り離すことで、セキュリティを損なわないようにするのも一つの方法です。また、フロントエンドからより多くのサービスにアクセスできるようになるため、アプリケーションのパフォーマンスが向上し、ユーザーエクスペリエンスが向上します。

6. サードパーティサービスの活用
サーバーレスは新しい分野であるため、ロギングやモニタリングなどの様々なサービスを提供する既存の企業向けツールには互換性がない場合があります。企業がサーバーレスのメリットを最大限に活用するためには、目の前のタスクを実行するための適切なサードパーティツールを選択することが鍵となります。

6.5.3コンプライアンス

継続的なニアタイムモニタリングを行い、脆弱性のドリフト、セキュリティやコンプライアンスのドリフトが発生したかどうかを検出し、記録します。

コンプライアンスを重視していない組織は、法的問題によって危機に陥る可能性があります。コンプライアンスのプロセスは非常に詳細であることが多く、広範な図や文書を作成し、組織のシステムがどのように準拠しているかについて監査人や規制機関に説明する必要があります。サーバーレスアプリケーションでは、セキュリティ管理の証拠として物理的なマシンを特定する機能がなくなるため、規制遵守が複雑になります。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

例えば、SOXのコンプライアンスを例に考えてみましょう。

強化されたソフトウェアエンジニアリングプラクティスとDevOpsポリシーは、SOX法コンプライアンスの大部分をカバーしています。しかし、完全なコンプライアンスを達成し、規制当局にそれを示すためには、個人を特定できる情報を扱う戦略を持つことが重要です。例えば、AWS Lambda機能の場合、SOXの重要な施行は、AWSの構成と開発パイプラインのポリシーとリソースへのコントロールになることを意味します。

同様に、GDPRでは、規制の重要な要素として、適切なセキュリティの実装、データへのアクセス制限、変更管理、データ保持、破棄のポリシー（一般的に組織がSOXに準拠するための取り組みの一環として実装される、上場企業に期待される全ての項目）が求められています。

7. サーバーレスセキュリティの未来像

これまで見てきたように、サーバーレスは多くのメリットと課題を伴っています。このセクションでは、運用、アプリケーションセキュリティ、暗号化/プライバシーの領域で、今後数年のうちにセキュリティにとって不可欠となる分野に焦点を当てます。サーバーレスのツールはまだ課題が多く、技術が成熟するにつれてCSPに縛られるようになります。ここ数年、クラウドプロバイダによるデプロイメントとコンフィギュレーションは改善されてきました。しかし、一部のクラウドプロバイダが他のプロバイダよりも成熟しているため、経験はまだ非常に断片的です。

本格的な導入には、利用者が複数のクラウド戦略を策定したり、CSPの上に抽象化レベルを設定したりするために、全体として同じレベルの成熟度が必要になります。また、クラウドプロバイダは、継続的なデリバリアプローチをサーバーレスに導入し、セキュリティ上の欠陥を検査・軽減するとともに、ABテスト、ブルー／グリーンデプロイ、カナリアリリースなど、デプロイ時の運用上のベストプラクティスをプラットフォームに組み込む必要があります。サーバーレスはカーディナリティが増加しているため、コンテナと同様の構成が必要になります。現在、それらのシステムをイントロスペクトするためのツールは、未解決の問題です。

運用上の課題と並んで、セキュリティの観点から見たアプリケーションの品質とその依存関係があります。アプリケーションのランタイム、サンドボックス、リスクを特定して軽減するためのAI/MLの手法などに焦点を当てたトピックは、サーバーレスの普及に伴い、今後数年間で増加するでしょう。開発者は通常、セキュリティに関するトレーニングを受けていません。そのため、斬新なプログラミング言語、より強固なSDLC統合、正式な手法により、開発者は適切なガードレールを使ってアジャイルに対応できるようになります。

最後に、サーバーレスの暗号化とプライバシーの面でもブレークスルーが必要です。ほとんどのCSPは、特殊なワークロードのためにsecure enclaveを採用していますが、これをサーバーレス製品にも拡張する必要があります。プライバシーの側面と、サーバーレスの一過性で際限なく広がる性質における暗号化されたデータの操作能力には、より優れたプライバシーフレームワークが必要であり、場合によっては暗号化されたデータの作り直しが必要となり、これは主要なクラウドプロバイダからいくつかのフレームワーク（SEAL）が発表されています。

サーバーレスが普及するにつれ、導入作業はCSPに成熟度と異なる環境での一貫性を求め、後押しすることになるでしょう。アプリケーションセキュリティのライフサイクルは、クラウドプロバイダとの間でより合理的に統合されるでしょう。また、DevOpsツールとサプライチェーンの統合は、特に最近のツールを利用したハッキング事件を受けて、大きな注目を集めている分野です。最後に、より遠大な目標であるにもかかわらず、プライバシーの側面ですが、主要なクラウドプロバイダはこの分野の研究開発を公開し、ロードマップとその焦点を明らかにしています。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

7.1 サーバーレスの未来への布石

7.1.1 サーバーレスへの動き コンテナイメージベースのサーバーレス

今日、すべての企業は、継続的な成長と競争力の強化のために、デジタルトランスフォーメーションを導入しています。

コンテナは現在、主にKubernetesオーケストレーションシステムを用いて広く展開されています。Kubernetesのようなクラウドネイティブテクノロジは、大規模なアプリケーション管理に必要な自動化、可視化、制御を提供し、高いイノベーションの速度を実現します。

傍目には、Kubernetesによる自動化によって、デプロイメントパイプラインやデプロイメント・運用プロセスにおける運用作業が最小限に抑えられているように見えるかもしれません。しかし、実際にはそうではありません。アプリケーションはKubernetesの構成で柔軟にデプロイできますが、多くの開発者にとってレプリカ、スケール、taint、トレランスを理解することは最重要課題ではありません。これは、開発者にとって不必要な複雑さにつながり、コンテナの未来がサーバーレスである理由を説明しています。

コンテナイメージベースのサーバーレスモデルでは、スケーリング、レプリカ、コントロールプレーンの管理など、多くの機能の管理をサービスプロバイダに抽象化します。これにより、開発者は自分のタスクに集中することができます。

今後は、コンテナイメージをベースにしたサーバーレスモデルがますます増え、開発者のオーバーヘッドを減らしながらも、開発者が行うべきことの柔軟性を高めていくことになるでしょう。

7.1.2 仮想化の変化

セキュリティはコンテナにとって大きな関心事です。仮想マシンはホスト上で非常に強力な隔離を提供します。しかし、コンテナは、Linuxの名前空間を使って、あるコンテナから別のコンテナへとリソースを分離、制限、隔離します。この隔離は侵害される可能性があり、コンテナはVMのようにセキュリティの境界として機能することはできません。高速なデプロイメント、簡単なコンポーネント評価、強力なセキュリティという機能の間で妥協する必要があるため、OSには様々なモデルが存在します。

Unikernel

Unikernelは、ライブラリOSを使用して構築された非汎用のもので、単一のアドレス空間のマシンを使用しています。このOSにより、攻撃対象やリソースフットプリントを大幅に削減することができます。

Kataコンテナ

Kataは、コンテナのように動作する軽量な仮想マシンを備えた安全なコンテナランタイムです。しかし、第二の防御層としてハードウェア仮想化技術を使用し、より強力なワークロードの分離を実現しています。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

gVisor

gVisorは、コンテナをサンドボックス化するための仮想化環境を提供し、セキュリティと隔離を高めます。通常はホストカーネルによって実装されるシステムインターフェースを、サンドボックスごとに独立したアプリケーションカーネルに移行することで、container escape exploit のリスクを最小限に抑えています。gVisorは、大きな固定オーバーヘッドを導入せず、リソースの使用に関してはプロセスライクなモデルを維持しています。

7.1.3 FaaSの進化

FaaSの導入が進むにつれ、企業のFaaSアプリケーションのデプロイメントに対して、より高い柔軟性、透明性、より強固なセキュリティ、コントロールを提供する必要性が高まっています。FaaSは、コンテナイメージをベースにしたサーバーレスプラットフォームの上に構築されます。

そのため、FaaSは以下のような様々な面で成長することができます：

Knative

Kubernetesが抽象化されたレイヤーとなる中、Knativeはオープンソースのプラットフォームであり、Kubernetesにサーバーレスのクラウドネイティブアプリケーションをデプロイ、実行、管理するためのコンポーネントを追加します。これにより、インフラ管理者は、既存のインフラをベースにしながら、プラットフォームの柔軟性と透明性を高めることができます。これにより、利用者はプライベートクラウド上でもファンクションを実行することができます。

OpenWhisk

FaaSの成長に伴い、ステートフルなFaaSをサポートすることが求められています。OpenWhiskは、イベントに応じて機能を実行する分散型サーバーレスプラットフォームをあらゆる規模で提供します。OpenWhiskは、Dockerコンテナを使用してインフラ、サーバーを管理するとともに、スケーリングを管理するため、利用者は素晴らしく効率的なアプリケーションの構築に集中することができます。OpenWhiskでは、Stateful Functionsが可能です。これにより、お客様はプライベートクラウド上でもファンクションを実行することができます。

OpenFaaS

OpenFaaSは、Knativeと同様に、DockerやKubernetesを使ってサーバーレスファンクションを構築するためのフレームワークであり、メトリクスを第一級にサポートしています。あらゆるプロセスを関数としてパッケージ化することができ、反復的な定型文のコーディングをすることなく、さまざまなWebイベントを利用することができます。

また、OpenWhiskと同様に、プライベートクラウド上で関数を実行することも可能になります。

Closed FaaS

Closed FaaSであっても、企業がFaaSのデプロイメントをコントロールできるように、関数を実行するコンテナ内で監視やセキュリティのロジックを実行できるようにする傾向が強まっています。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

7.2 サーバーレスのセキュリティ

「サーバーレスアーキテクチャをいつ、どのように使用するかについての理解は、まだ初期段階にあります。推奨されるプラクティスのパターンが発生し始めており、この知識は今後も増えていくでしょう」
(Martin Fowler)

まだ初期段階にあるサーバーレスの、実用的な知恵と課題を示す言葉が上記です。プラクティスとアーキテクチャパターンは、今後数年間、開発、デプロイ、モニタリング、パフォーマンスの向上、よりセキュアなアプリケーションの提供などに最適なアプローチを学ぶために成長し続けるでしょう。

1. **サーバーレスは、開発者に焦点を当てたもので**、コードファーストです。現在、クラウドプロバイダ間での言語のサポートには変化と課題があります。**フレームワークを利用することで解決できます。**サーバーレスアプリケーションの設定やデプロイに必要なタスクの中には、サーバーレスフレームワークの助けを借りずに行うには時間がかかりすぎるものがあります。開発者がこれらの作業を手作業で行っても何のメリットもありませんので、フレームワークを活用することは非常に理にかなっています。**フレームワークは今後も進化し続けます。言語のサポート強化、継続的な開発と統合、デプロイメントのためのコードのバンドルとパッケージ化、実際のデプロイメントプロセス（それ自体が複数のステップを要する場合もある）、環境変数の設定、シークレットの管理、サーバーレスサービスをパブリックAPIとして公開するためのエンドポイントの設定、関数に必要なパーミッションの取得と適切な設定など。**

現在、サーバーレスフレームワークのエコシステムは非常に豊富で多様です。あるものは、特定のプログラミング言語やクラウドプロバイダに焦点を当てています。また、ランタイムやクラウドに依存しないものもあります。今後は、よりクラウドネイティブに対応し、開発言語のサポートを強化し、クラウドにとらわれないものにシフトしていくのがよいでしょう。

2. **規格は進化して**おり、サーバーレス環境における**セキュリティ制御の調和**に向けた取り組みが行われています。

3. **ロギングとデバッグ** – 開発者は、コードを実行しているプラットフォームをコントロールする必要があり、バックグラウンドプロセスやデーモンによって生成される信頼できるメトリクスが必要です。コードをインストルメント化して、メトリクスをサードパーティのサービスにリアルタイムで送信することもできますが、それはレイテンシーが全体の実行に影響することを意味します。サーバーレスアプリケーションへのログインには特定の情報が不可欠であり、セキュリティ侵害に対処する時に利用できるようにしておく必要があります。重要なログがあれば、例えば、攻撃者がどのセキュリティの欠陥を探ったのか、それをどのように修正するのかを理解したり、IPアドレスのブラックリストを構築したり、侵害された顧客アカウントを特定したりするのに役立ちます。

サーバーレスアプリケーションのロギングとデバッグを適切に行うためには、これらの活動へのアプローチ方法を再考する必要があります。必要とされるパラメータ、例えば呼び出し/イベントのインプット、応答ペイロード、認証要求などへのアクセスと、オープンインテグレーション、完全性のアベイラビリティは進化する必要があります。

4. **ステートフルとトレースへの移行** – サーバーレスの機能は一時的なもので、本質的にデータの保持ができないことから、ほとんどの場合、外部のサービスと相互作用します。

© Copyright 2021, Cloud Security Alliance. All rights reserved.

データベースであれ、オブジェクトストレージであれ、データレイクであれ、データのステートフル性に依存したタスクを遂行するために、関数は外部ストレージを必要とします。関数と相互作用する他のサービスには、メッセージキュー、データストリーミング処理、認証メカニズム、機械学習モデルなどがあります。

サーバーレス関数のライフサイクル全体を追跡する一方で、パフォーマンスの向上、セキュリティの監視、デバッグなどのためには、ランタイムの実行が不可欠です。こうした外部とのやりとりはすべて、そのための困難をもたらします。インストルメンテーションのためのトレースシステムを選択する際に考慮すべき重要な点は、そのスケーラビリティと可用性です。サーバーレスプラットフォームは、定義上、非常に迅速にスケーリングでき、高い可用性を提供できるため、トレーシングシステムは、サーバーレス関数の弾力的な要求に対応できなければなりません。このため、サーバーレス関数のインストルメントに特化したソリューションは進化する必要があります。

5. **パフォーマンス**：サーバーレス関数は、スケーラビリティの面で実質的に無限の弾力性を備えていますが、監視をせずに稼働させていいということではありません。期待されるパフォーマンスのしきい値を設定し、何か注意が必要なときに判断できるようにすることが重要です。
注目すべき重要な指標は以下の通りです。

- インボケーション呼び出し回数
- ランタイムのクラッシュ、アプリケーションの失敗、コールドスタート、再試行の回数
- メモリ使用量
- 実行時間

コールドスタートは、ある時点でリクエストの数に対応できるだけの十分なコンテナが関数にない場合に起こります。これにより、基盤となるサーバーレスプラットフォームは、リクエスターがレスポンスを待っている間に、数百ミリ秒から数秒かかるかもしれない新しいコンテナのスピンアップを余儀なくされます。これが望ましくないシナリオはたくさんあります。アプリケーションがこのような状況にある場合、スタック内のコールドスタートを検出して監視する必要があります。クラウドサービスは通常、この情報を直接提供しませんが、監視サービスは進化する必要があります。

6. **IAM** サーバーレス関数は、ドメイン内のサービスと統合することができ、潜在的に信頼ドメインを越えることができます。このため、コンテナイメージベースのサーバーレスやFaaSでは、トラスト管理やアクセス管理が複雑になります。プラットフォームやドメインを超えて、統一されたサービスの信頼とユーザーのアイデンティティを利用できることは、サーバーレスの利用が拡大する中で非常に重要です。業界では、クロスプラットフォームの認証とアクセス管理のために、SPIFFEとSPIFFE Runtime Environment（SPIRE）の開発が進められています。SPIFFEはまだ正式な規格ではありませんが、批准される過程にあります。SPIFFE/SPIREは、観測性、監視、そして最終的にはサービスレベル目標（SLO）の向上にも役立ちます。SPIFFE/SPIREは、様々なシステム（必ずしもコンテナ化されたものやクラウドネイティブなものだけではない）におけるソフトウェアのアイデンティティを正規化し、アイデンティティの発行と使用に関する監査証跡を提供することで、インシデント発生前、発生中、発生後の状況認識を大幅に改善することができる。より成熟したチームであれば、サービスの可用性に影響を与える前に問題を予測する能力が向上することもあるでしょう。

7. **サプライチェーンの**開発者は、その関数の様々なライブラリを使用することがあります。ライブラリを使用することで、開発者は既存の機能を記述するのではなく、利用したり再利用したりすることで時間を節約することができます。これらのライブラリは、サードパーティによって開発されたり、オープンソースであったりします。通常、

© Copyright 2021, Cloud Security Alliance. All rights reserved.

ライブラリが効率的で脆弱性のないものであるという保証はありません。そのライブラリの開発者は、他のライブラリを使用している可能性があります。ライブラリの数は、そのサプライチェーンの数層分の深さがあるかもしれません。

コードの一部が実行するロジックでライブラリを使用する場合、そのライブラリは技術的には依存関係にあります。コードの層は、コードの中に依存関係として導入されます。依存関係が他の依存関係を使用すると、これらの層はより深くなります。脆弱性はどのレイヤーやブランチにも存在する可能性があり、深刻度や悪用の難易度に応じて、サーバーレス機能に影響を与える可能性があります。そのため、依存関係のツリーを理解し、できるだけ短く、狭くすることが不可欠です。依存関係に脆弱性がないか定期的にチェックすることは、アプリケーションのセキュリティ態勢を維持するための良い習慣です。また、Node.JSや最近のSolarigateの侵害など、一連の漏洩事件を受けて、クラウドネイティブな技術や機能のサプライチェーンセキュリティについても多くの取り組みがなされています。これらのベストプラクティスは、サプライチェーン・ソフトウェアからの依存関係に信頼を確立するためのメカニズムを提供し、サードパーティのライブラリやオープンソース・ソフトウェアからの依存関係にある脆弱性からサーバーレス機能を安全に保つことができます。

サーバーレスは他の技術に比べてまだ初期段階にあり、アプリケーションの開発、デプロイ、監視をより良く行うための最適なアプローチについては、今後数年で知識が増えていくと思われます。(Martin Fowler)

7.3 データプライバシーのためのサーバーレスの進歩

サーバーレスは、組織がクラウドネイティブ機能を構築し、利用し、統合する方法を形成する最大の原動力になると考えられているため、（クラウドプロバイダがスティッキネスを高めたいと考え、BaaSの提供を増やしていることもあり）データプライバシーを保護するためのメカニズムが必要です。

この1年で、著名なベンダーによる機密性の高いコンピューティング[2]が増えてきました。
例：（AWS Nitro、Azure confidential computing、GCP confidential computing）。これらの機能の一部はサーバーレスにも移植される傾向にあります[Gartner, 2021]。

より多くのワークロードがサーバーレスに移行すると、データ、ステートデータ、メタデータを暗号化して処理することが必要になります。いくつかの例を見てみると、おそらくCSPのサーバーレスサービスに組み込まれるでしょう。

自己防衛: 関数はリアルタイムで、各リソースのセキュリティとマイクロセグメンテーションを評価し、適応させます。さらに、予測分析（AI/ML）を使用してセキュリティポスチャを評価し、新しい警告メカニズムを定義します。

予め定義されたコンプライアンス関数：設定されたガードレールや必要なデータの種類に応じて、開発者が活用・利用できる関数のセット。これは、ガードレールを設定するための自然な延長線上にあり、サーバーレスのカタログのようなものです。これは、アジリティを低下させることなくガードレールを作ることができるというメリットがあります。

[2] “By 2025, 50% of large organizations will adopt privacy-enhancing computation for processing data in untrusted environments and multiparty data analytics use cases.” - Gartner.

© Copyright 2021, Cloud Security Alliance. All rights reserved.

FaaSのためのsecure enclave：規制対象となる業界では、クラウド環境の精査と保証が求められており、企業は増え続ける規制に準拠するための仕組みを活用する必要があります。

データ、コード、計算の機密性、完全性を維持するために、信頼できる実行環境が広く利用されていることを見てきました。その一例として、世界的に関連のある欧州一般データ保護規則（EU GDPR）が挙げられます。これらのメカニズムは、GDPR対応のクラウド環境、中でも従来型のIaaS環境でFaas関数がより利用されるようになると、有益となるでしょう。

この技術は、コード、データ、処理を他のお客様やCSPからよりよく分離することができます。個人情報や機密データが含まれている場合は特に重要です。

すでに、**enclave**をサービスとして提供しているライブラリ（IEEE Xplore[3]、Graphene[4]など）も存在しており、自然な流れでFaaSへの展開も視野に入れています。

8. 結論

ほぼすべての業界のIT組織は、価値をより速く提供し、競争に先んじて、顧客に新しい体験を迅速に提供しなければならないというプレッシャーを感じています。サーバーレスプラットフォームは、アプリケーションチームが、アプリケーションが動作するインフラを管理することなく、価値を提供することを可能にします。この動きが活発になるにつれて、サーバーレスプラットフォームが普及し、これらのプラットフォームに高価値のアプリケーションが置かれるようになるでしょう。サーバーレスプラットフォームにおけるセキュリティの懸念は、ここからさらに大きくなっていくでしょう。

CSA Serverless Working Group として、私たちは、コンテナイメージベースのサーバーレスとFaaSの両方のプラットフォームが現在存在しており、今後進化していくことを想定して、サーバーレスセキュリティのあらゆる側面を捉えました。今後、サーバーレスプラットフォームに大きな変化があった場合には、この文書を新たに改訂する予定です。

[3] https://ieeexplore.ieee.org/document/7163017
[4] https://grapheneproject.io/

© Copyright 2021, Cloud Security Alliance. All rights reserved.

Appendix 1: Acronyms

ABAC	Attribute-based access control
API	Application Program Interface
BaaS	Backend as a Service
CICD	Continuous integration, continuous delivery
CISO	Chief Information Security Officer
CPU	Central processing unit
CRUD	Create, Read, Update, Delete
CSP	Cloud Service Provider
DevOps	A portmanteau of "development" and "operations."
DLP	Data Loss Prevention
FaaS	Function as a Service
GDPR	General Data Protection Regulation
HIPAA	The Health Insurance Portability and Accountability Act of 1996
IaaS	Infrastructure as a Service
IT	Information Technology
KMS	Key Management System
NIST	National Institute of Standards and Technology
OPA	OPA (Open Policy Agent) is an open-source, general-purpose policy engine that unifies policy enforcement across the stack. OPA provides a high-level declarative language that lets you specify policy as code and simple APIs to offload policy decision-making from your software.
OS	Operating System
PaaS	Platform as a Service
PCI	Payment Card Industry
SLO	Service Level Objectives
SPIFFE	Secure Production Identity Framework for Everyone
SPIRE	SPIFFE Runtime Environment
STS	Security Token Service
VNet	Virtual Network
VPC	Virtual Private Cloud

© Copyright 2021, Cloud Security Alliance. All rights reserved.

Appendix 2: Glossary

Service boundaries	Service boundaries are defined by the declarative description of the functionality provided by the service. A service - within its boundary - owns, encapsulates and protects its private data and only chooses to expose certain(business) functions outside the boundary.

© Copyright 2021, Cloud Security Alliance. All rights reserved.