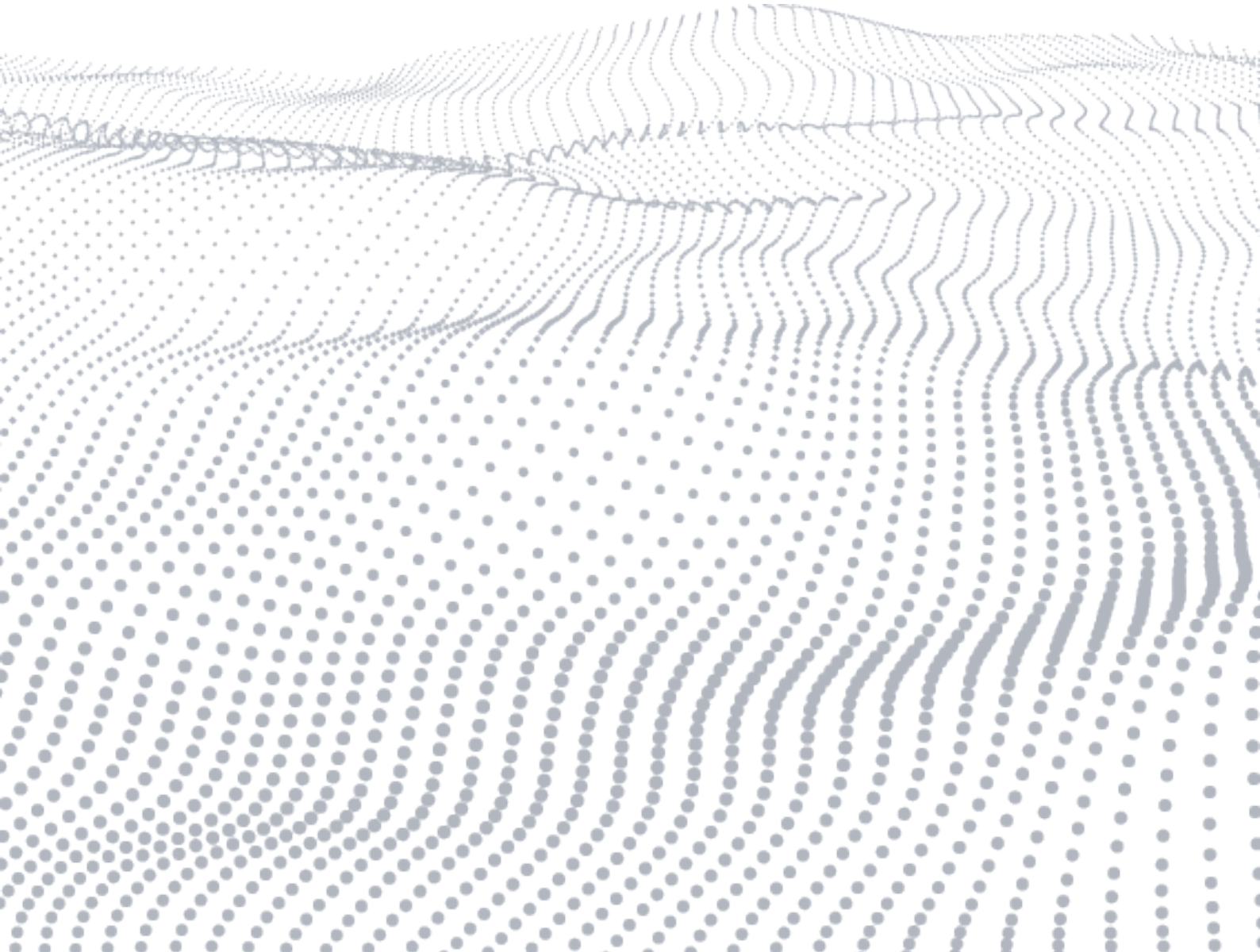
BLACKDUCK®

ホワイトペーパー

エージェント・インストルメンテーションを使用したソフトウェア定義自動車のファジング・テスト

Rikke Kuipers、 Dennis Kengo Oka（ブラック・ダック）



概要

この数年間で、サイバー・セキュリティが自動車開発プロセスの各ステップに結び付けられるようになりました。特に、ファジング・テストは自動車システムで未知の脆弱性を検出するための強力な手法であることが証明されています。ただし、インストルメンテーションが限られている場合、特に高性能コンピューター（HPC）などソフトウェア中心のシステムでは、メモリ・リークやアプリケーションがクラッシュしてもすぐに再起動するケースなど、いくつかのタイプの問題が検出されません。

これらの自動車システムは Linux や Android などのオペレーティング・システムがベースであるため、被試験システム（SUT）から情報を収集して、ファジング・テスト中に例外が検出されたかどうかを判定することができます。検出された例外に関するこれらの詳細により、開発者は問題の根本原因を的確に理解および特定し、問題をより効率的に修正することができます。

このホワイトペーパーでは、Agent Instrumentation Framework を紹介し、それを使用してどのように HPC のファジング・テストを改善できるかを説明します。検出された問題の根本原因を開発者が突き止めることができるように、SUT 上で例外を特定するためにターゲット・システムからどのように情報を収集できるかを説明します。また、この手法と、複数の SUT に対して実行されたファジング・テストに基づいて、テスト・ベンチを構築します。結果に基づいて、Agent Instrumentation Framework がなかったら検出されなかった可能性がある問題の例をいくつか挙げます。

はじめに

自動車産業のドライビング・イノベーションには、コネクテッド（Connected）、自動運転（Autonomous）、シェアリング（Shared/Services）、電動化（Electric）の 4 つの主な動向（CASE）があります。これらの動向によって安全性、快適性、ユーザー・エクスペリエンスが向上しますが、車両内で使用されているソフトウェアの脆弱性の可能性が高まり、攻撃対象領域が広がり、車両内がより価値のあるターゲットになります。そのため、サイバー・セキュリティのニーズが高まっています。自動車業界は、サイバー・セキュリティの規格と規則に対応しています。たとえば、2021 年 8 月に『ISO/SAE 21434:2021 自動車ーサイバー・セキュリティ・エンジニアリング（Road vehicles—Cybersecurity engineering）』、2021 年 2 月に『Automotive SPICE for Cybersecurity』が公開され、2021 年 1 月に UNR155 – サイバー・セキュリティおよびサイバー・セキュリティ管理システムが施行されました。

V モデルとして一般に知られている標準的な自動車開発プロセスでは、すべてのステップでサイバー・セキュリティが不可欠な要素です。セキュリティ・アクティビティには、セキュリティ要件の定義とレビュー、設計のセキュリティ・レビューの実施、ソフトウェア開発中のベスト・プラクティスと安全なコーディング規格の適用、静的コード解析ツールなどの自動化ツールの使用、機能テスト、脆弱性スキャン、ファジング・テスト、ペネトレーション・テストなどのテスト・アクティビティの実行などがあります。[1]

具体的には、自動車セキュリティ・テストに関して、『Security Crash Test—Practical Security Evaluations of Automotive Onboard IT Components』では、さまざまなテスト手法の総合的な概要が示されています。[2] きわめて強力なテスト手法の 1 つに、ファジング・テストがあります。これにより、開発者とテスターはシステムおよびコンポーネント内の未知の脆弱性を特定することができます。ファジング・テストは、被試験システム（SUT）に対して不正入力または「規格外」入力が行われた後、例外または意図しない挙動があるかどうかを検出するテスト手法のタイプです。

現在、コネクテッド・カーと自動運転の領域での開発が増えています。ソフトウェア定義車両への移行も進んでいます。つまり、多数の電子制御ユニット（ECU）に分散されていた車両機能が少数の高性能コンピューター（HPC）に統合されています。これらはソフトウェア中心のソリューションであるため、それらのソリューションに AUTOSAR Adaptive Platform や Linux/Android などの複数のオペレーティング・システムを含めることができます。それらのソリューションでは、自動運転、インフォテインメント、デジタル・コックピット、ゲートウェイ、コネクティビティ・ユニットを提供するなど、多くの異なるアプリケーションを実行できます。これらのアプリケーションの一部は、車両の外部と相互作用するため、攻撃者にとって非常に魅力的な標的となります。車両またはシステム・メーカーの管理外のシステムまたは環境から異常入力や不正入力も行われやすくなります。そのため、セキュリティ、堅牢性、安全性を確保するために、これらのタイプのシステムのファジング・テストを実行することがきわめて重要です。

いくつかの自動化ツールでは、自動車システムでファジング・テストを実行することができます。すなわち、SUT と通信するための関連プロトコルがサポートされています。そのため、ファジング・テストを簡単に実行するには、帯域内インストルメンテーションを使用してファジング対象の通信チャネルを監視します。ただし、この手法の課題の 1 つとして、SUT のインストルメンテーションを行って、SUT 上に例外があったかどうかや SUT で障害が発生して SUT がクラッシュしたかどうかを判定するのが難しいことがよくあります。さらに、テストは一般的にブラックボックスとして実行されるため、SUT から十分な情報を収集して例外または障害の根本原因を特定するのが難しくなります。この情報は、システムの開発者が問題を迅速かつ的確に見つけて修正するのに役立ちます。

以前の調査で、HIL（Hardware-In-the-Loop）システムを使用したインストルメンテーションによって ECU に対し効率的にファジング・テストを実行することができることが示されています。[3,4] その調査で提示されたソリューションでは、HIL システムで ECU によって生成されたアナログおよびデジタル信号を測定することで被試験 ECU の挙動を監視する方法が示されています。この統合ソリューションで

は、HIL システムが SUT 上に例外があるかどうかを判定するさまざまな方法が提供され、ファジング対象プロトコルのみが監視されていた場合には検出できない例外を検出することができます。たとえば、CAN（Controller Area Network）バスがファジング対象の場合、ターゲット ECU が誤動作し、アナログまたはデジタル出力を生成して誤ってアクチュエーターを制御しようとすることがあります。CAN バスのみが観察されていた場合は、この意図しない挙動が見逃されます。HIL システムを使用して ECU のインストルメンテーションを行うことで、この異常な動作を検出することができます。

一方、このホワイトペーパーでは、ファジング・テストのソフトウェア定義車両、具体的にはインフォテインメント・システムや接続装置など、外部から操作する HPC に注目します。このようなソフトウェア中心のシステムに対し的確にインストルメンテーションを行い、効率的で正確なファジング・テストを実行することができる Agent Instrumentation Framework を紹介します。すなわち、これらのタイプのシステムに対して、インストルメンテーションを改善する追加手法を取ることができます。これらのシステムは Linux や Android などのオペレーティング・システムがベースであるため[5,6,7]、SUT から情報を収集してファジング・テスト中に例外が検出されたかどうかを判定することができます。さらに、検出された例外に関する詳細をファジング・テスト・ツールに返し、ログ・ファイルに保存することができます。この追加情報により、開発者は根本原因を的確に理解および特定し、最終的に問題を解決することができます。

背景と問題の説明

多くの自動車業界は、ファジング・テストをソフトウェア開発プロセスでの必須ステップにしているか、その方向に移行しています。通信業界などでは、ファジング・テストはすでにソフトウェア開発プロセスに統合され、バグや脆弱性を迅速に特定する効果的な手法であることが証明されています。これらのターゲット・システムは、ファジング対象の同じプロトコルを監視することで、一般的に実装しやすくなっています。これは、web サーバーや特定の通信ライブラリなどの IT ソリューションをファジングする際の一般的な手法です。

多くの場合、ファジング対象の同じプロトコルを監視すると効果的です。たとえば、HTTP リクエストと対応する HTTP レスポンスを監視すると、web サーバーの未知の潜在的な脆弱性を特定できます。さらに、OpenSSL ライブラリをファジングすることで見つかった、有名な脆弱性であるハートブリード（CVE-2014-0160）は、Heartbeat Request メッセージに対するレスポンスを監視することで特定されました。[8] 一方、自動車システムは複雑で他のシステムと相互接続されていることがよくあるため、実装は簡単ではありません。結果的に、適切なインストルメンテーションがなければ、多くの未知の脆弱性や潜在的な問題をこれらのシステム上で特定できません。

しかし、前述のとおり、HIL システムを使用して深く組み込まれた ECU の適切なインストルメンテーションがなかったら、いくつかの潜在的な問題が検出されません。同様に、HPC の適切なインストルメンテーションがなかったら、多くの潜在的な問題が検出されません。

一般的に、接続装置で Wi-Fi や Bluetooth によるファジング・テストなど、特定のプロトコルのファジング・テストを実行する際には、ファジング対象の同じプロトコルによってインストルメンテーションが実行されます。つまり、SUT はファジング対象の同じプロトコルによって監視されます。図 1 に示すように、この制限された帯域内インストルメンテーションによっていくつかの問題が検出されない可能性があります。

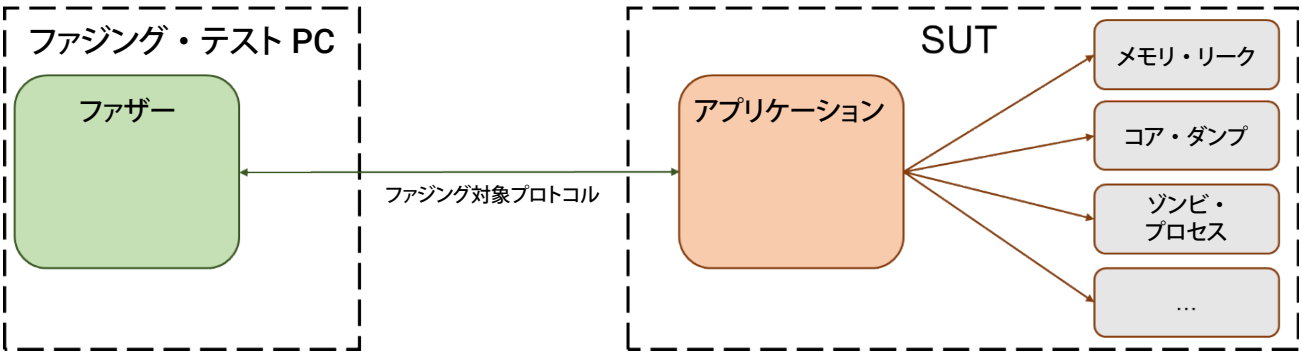


図 1：ファジング対象プロトコルによって検出されない SUT 上の問題の例

ファジング・テスト中に検出できない問題の 1 つは、メモリ・リークです。ファジング・テスト中に、ファジング対象プロトコルによって送信されたメッセージが SUT によって処理されます。この例では、SUT によってファジング対象メッセージが正しく処理され、ファジング対象プロトコルのみのインストルメンテーションを行うことで、問題の兆候は見られません。ただし、ファジング対象メッセージの処理により、メッセージを処理するアプリケーションでメモリ・リークが発生します。特定のタイプのメッセージに対して十分な時間ファジング・テストが実行されている場合は、ファジング対象プロトコルによって例外を検出できることがあります。たとえば、レスポンスに予想よりも時間がかかる、レスポンスがないなどです。ただし、プロトコルによる全通信メッセージの解析に膨大な時間を費や

さなかったら、この挙動の原因を特定することは通常きわめて難しいでしょう。一方、メモリ・リークの原因となった最初のファジング対象メッセージの後に SUT 上でメモリ・リークを直接検出することは非常に効率的であり、開発者が原因を正確に特定することができます。

ファジング・テスト中に見落とされる可能性のあるもう 1 つの問題として、特定のファジング対象メッセージによってアプリケーションがクラッシュする場合があります。ただし、通信プロトコルによってインストルメンテーションが実行されるまでにアプリケーションはすぐに再起動し、正しく応答しているように見え、例外（クラッシュ）は正しく特定されません。機能上、ユーザーからはすべてがあるべき動作をしているように見える（ユーザーがクラッシュに気付かないほどの速さでアプリケーションが起動して動作する）ため、この挙動は許容される場合があります。ただし、この例外は時間の経過とともに SUT に悪影響を与える場合があります。最終的にシステムをクラッシュさせたり意図しない方法で動作させたりする、ゾンビ・プロセスやコア・ダンプなどです。また、クラッシュを発生させたメッセージを攻撃者が特定できる場合もあります。脆弱性を悪用できる場合、攻撃者はリモート・コードを実行できる特定のメッセージを巧妙に作り上げることができます。アプリケーションはクラッシュしないため、再起動もされず、攻撃者は完全に制御できます。

これらの例の問題では、脆弱性やバグが検出されない可能性があります。ファジング・テストは通常ブラックボックス手法ですが、開発者やテスターは SUT（Linux や Android など）の内部にアクセスできる場合があります。これにより、例外を特定できるスクリプトを実行できます。すなわち、SUT 上に配置されたエージェントで外部インストルメンテーションによって監視を支援するグレー / ホワイト・ボックス手法が採用されることで、より効率的で正確なファジング・テストが実現されます。この手法については、次のセクションで詳しく説明します。

Agent Instrumentation Framework の概要

Agent Instrumentation Framework では、ファザーに SUT の詳細なインストルメンテーション・データが提供されます。このデータを使用して、テスト・ケースの失敗 / 成功の判定を行うことができます。また、このデータは SUT の有益な情報を含むため、テスターが、見つかった問題を解決するために使用することができます。

同じ SUT は二つとないため、Agent Instrumentation Framework をモジュール方式にし、その機能の迅速な作成と適応が可能であるように設計しました。テスト対象システム上で実行されているソフトウェア・モジュールはエージェントと呼ばれます。エージェントの主な目的は、インストルメンテーション・タスクを 1 つ実行し、この収集された情報をファザーに返すことです。ただし、ファジング・プロセスを自動化し、より複雑で詳細なインストルメンテーションを可能にするために、テスト・ケース実行時の他の時間帯にエージェントで機能が実行される場合もあります。

重要なのは、エージェントがインストルメンテーション・タスクを 1 つだけ実行することです。エージェントが判定基準として収集するデータが多すぎる場合、フレームワークを使用するテスターはログ・ファイルを検索して障害の理由を詳しく検討する必要があります。さらに、すべての SUT が実装可能な同じ機能を備えているわけではありません。エージェントがさまざまな構成やシナリオと互換性があるようにすると、テストが遅れ、エージェントを常時変更しなければならない可能性があります。

ブラック・ダックは、それぞれ構成が異なる複数のエージェントを同時かつ自動的に設定、起動、制御する機能を持つフレームワークを設計しました。この柔軟なモジュール方式の手法では、すべてが SUT 上で異なるパラメータを使用して実行されている、複数の（個別に作成された）エージェントを使用できます。

要件、アーキテクチャ、構成

Agent Instrumentation Framework の主な目的は、より高度なインストルメンテーションを使用することで未知の脆弱性を見つけることです。そのため、車載インフォテインメント（IVI）システムと接続装置をテストする際に、ブラックボックス手法から若干離れることは避けられません。ユーザーが求める脆弱性のタイプによっては、エージェントがそのタスクを正しく実行できるように、ユーザーは SUT 上で実行されているオペレーティング・システムにアクセスしなければならない場合があります。

組み込みデバイスのセキュリティに関する多くの文書で、IVI または接続装置のオペレーティング・システムへのアクセスは大したタスクでないことが明らかになっています。OS へのリモート接続は、内蔵ワイヤレス通信テクノロジー、シリアル、または USB からイーサネットへのコンバータを介して行うことができます。オペレーティング・システムにアクセスしてオペレーティング・システムとの通信を設定する方法については、この文書では触れません。

Agent Instrumentation Framework は、一般的な Golang プログラミング言語で記述されています。Golang はエージェントの書き込みにおける柔軟性が高いライブラリが充実しています。さらに、Golang は静的に（クロス）コンパイルできるため、仮想組み込みアーキテクチャ上でフレームワークを実行できます。フレームワークは、x86、AMD64、ARM、ARM64 のバイナリを提供します。

フレームワークの注目点は、精度が向上し、インストルメンテーション機能が拡張されることです。これにより、ファザーは 1 つのテスト・ケースに対してイベントを正確に特定することができます。このため、フレームワークに含まれているエージェントはすべて、テスト・ケースの実行に同期して運用されます。

エージェントは、ファジング・セッションに従って実行されます。これにより、エージェントの機能の速度に応じて、実行される各テスト・ケースにオーバーヘッドが加えられます。ユーザーは SUT のハイレベル障害表示を検知するため、最初のテストケース群は少数エージェントの設定で実行することができます。その後、より多くのエージェントを有効にし、障害の原因を一つに絞り込むことができます。そのため、全体的な有効性とファジング・セッションの持続期間には、エージェントのタスクを最小限かつ的確に保持することが不可欠です。

インフォテインメント・システムでのバッファ・オーバーフローを考慮し、1,000 テスト・ケースを超える測定可能な障害を徐々に作り上げます。プロトコル固有のインストルメンテーションのあるファザーを使用すると、実際に障害が発生したときに、テスト・ケース 1,000 で初めて障害の兆候を示すことができます。一方、同期して実行されるエージェントは、原因がテスト・ケース 143、376、1,000 などであると判定する可能性があります（テスト・ケース 143 と 376 はある種の例外を示しますが、完全な障害は発生しません）。

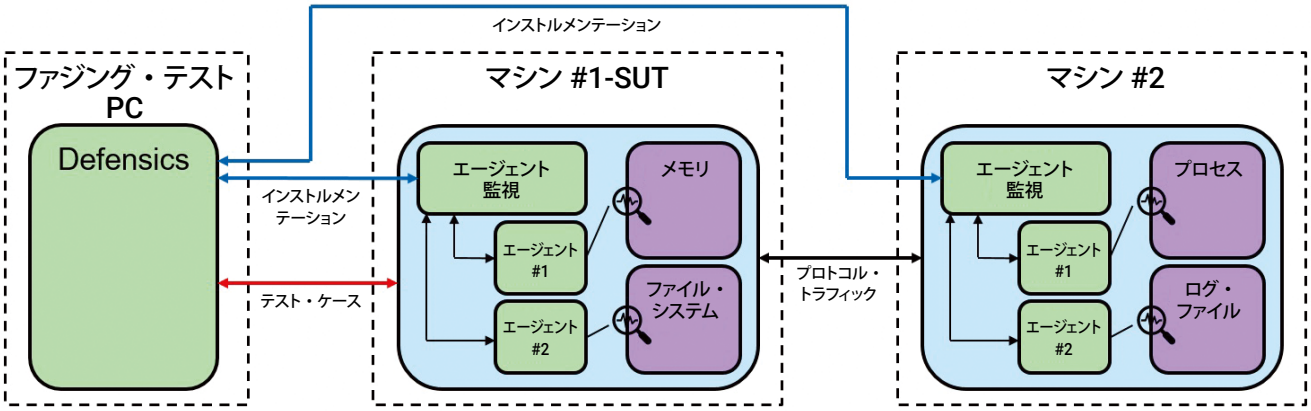


図 2：エージェント・フレームワークのアーキテクチャ

エージェント監視のデプロイメント

Agent Instrumentation Framework は、さまざまなプラットフォーム・アーキテクチャの 1 つのバイナリとして提供されています。このバイナリは、実行時に、サーバーとして機能し、選択したファジング・プラットフォーム、ブラック・ダックの Defensics® ファザーから受信した接続要求をリッスンします。Defensics では、すでに、プロトコル固有の RFC 準拠シーケンス、SNMP、syslog、機能プロトコル・チェックなど、デフォルトで提供されているインストルメンテーション機能をユーザーが簡単に拡張できるようになっています。フレームワークでは、それらの機能がさらに拡張されます。

監視サーバーは、web サーバーに REST API を実装します。監視サーバーは、`server` 引数を指定することで起動します。Defensics とフレームワークの間のトラフィックは、以下のオプションによってセキュリティ保護されます。

- HTTP サーバーが任意のポートで実行される。
- HTTP が TLS によって常に保護されている。
- HTTP が自己署名 X.509 証明書によって保護されている。この証明書は、監視サーバーの起動時にコンソール引数としてクライアントに提供される。
- HTTP がベアラー・トークンによって認証される。ベアラー・トークンは、コンソール引数としてクライアントに提供される。

監視サーバーにはいくつかの構成オプションが用意されています。これについては、`--help` 引数を使用してサーバーを実行する際にヘルプ・テストで詳述されます。これらのオプションは、ログ記録、暗号化、CLI 構成の制御を強化できるほか、プロジェクトのライセンス使用方法についての情報を提供します。

```
p0c@h4x0rz ~ $ /opt/Synopsys/Defensics/monitor/agent/agent_linux_amd64
Defensics Agent Instrumentation server provides a RESTful HTTP API for communicating with instrumentation Agents.

See Defensics User Guide for more information on how to connect to the server from Defensics.

Exit codes for different situations:

0: Regular exit without errors
1: Generic error
2: Generic error in server startup
3: Port in use
4: Certificates not available/readable
5: Invalid token provided

Usage:
  agent_linux_amd64 [command]

Available Commands:
  completion    Generate shell completions
  generateCerts Generate certs for HTTP API
  help          Help about any command
  listLicenses  List licenses for third party components
  server        Run Agent server
  version       Prints Agent Framework version

Flags:
      --config string      Config file (default "/home/p0c/.synopsys/agent-instrumentation/agent-instrumentation.yaml")
  -h, --help               help for agent_linux_amd64
  -f, --logformat string   Logging format: text, json. (default "text")
  -l, --loglevel string    Logging level: ERROR, WARN, INFO, DEBUG, TRACE. (default "INFO")
  -w, --workdir string     Work directory for storing data generated during testing (logs etc) (default "/home/p0c/synopsys/agent-instrumentation")

Use "agent_linux_amd64 [command] --help" for more information about a command.
```

図 3：エージェント監視の構成オプション

エージェント監視サーバーは、以下の方法でデプロイできます。

- デプロイメント・オプション 1：ファザーと同じシステム上。これは、ファザーと SUT が同じマシン上にあるか、エージェント監視によって作成されたエージェントが SUT 上に存在する必要がないことを意味している可能性があります。エージェントは、エージェントの論理によって行われた判定についての情報を提供する API と相互作用する可能性があります。
- デプロイメント・オプション 2：SUT と同じシステム上。エージェント監視によって作成されたエージェントは、ファイル・システム、デーモン、またはその他の OS 構造や情報ソースとの相互作用を必要とすることが多いため、これは一般的なデプロイメントです。
- デプロイメント・オプション 3：SUT 以外のシステム上。多くの場合、テスト・ケースのコンテンツが SUT 以外のアプリケーションやテクノロジーに影響を与えます。複数のエージェント監視を異なる場所で開始でき、すべてがファジング・セッション中に Defensics に情報を返します。

エージェントの作成

サーバー引数を指定してバイナリを呼び出すことでエージェント監視が開始されると、Defensics がそれに接続して 1 つまたは複数のエージェントを動的にデプロイおよび設定できるようになります。エージェントのデプロイと設定は自動的に開始され、エージェント監視からの指示に基づいてその機能が実行されます。デプロイメントに応じて、2 つの構成が可能です。

- ローカル・エージェント・サーバーを自動的に起動：デプロイメント・オプション 1。
- 起動したサーバーに手動で接続：デプロイメント・オプション 2 および 3。

2 つ目の構成では、エージェント監視によって `generateCerts` 引数を介して CA 証明書と認証トークンが生成される必要があります。また、引数として `--insecure` を指定することで、安全でない接続を使用できます。

[**Fetch available agents**] ボタンを押すと、使用可能なすべてのエージェント・タイプのエージェント監視に対してクエリーが実行されます。独自のカスタム・エージェントを作成してエージェント監視バイナリにコンパイルすることもできます。ドロップダウン・リストからエージェントを選択すると、指定した構成でエージェントが作成されます。

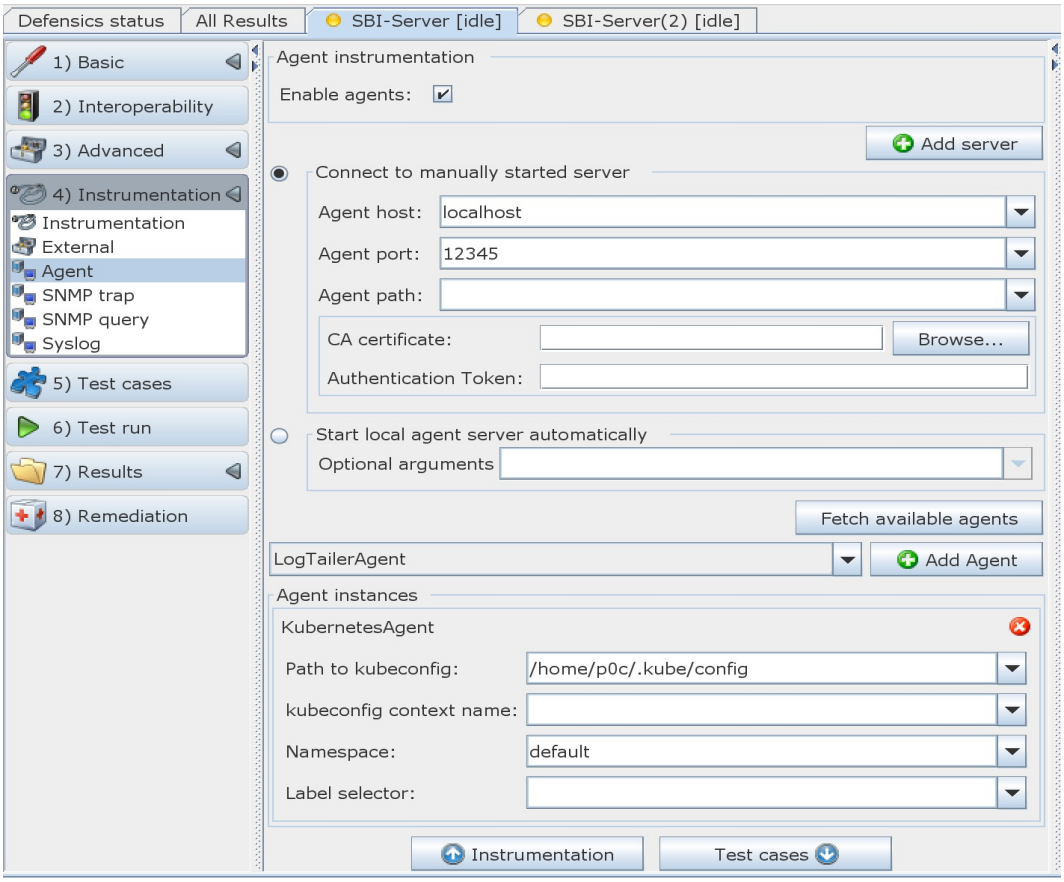


図 4：Defensics でのエージェント監視の構成

カスタム・エージェント開発

Agent Instrumentation Framework には、デフォルトでいくつかのエージェントが含まれていて、ファザーにあるデフォルトのプロトコル固有のインストルメンテーションを拡張します。インストルメンテーションのデフォルト・タイプは、通常、全テスト・ケースのメッセージ、または SUT が動作していることを確認する基本 ICMP メッセージの有効な RFC 準拠シーケンスの実行を必要とします。

独自のエージェントをプラグインとして実装することで、それらのエージェントをフレームワークに追加することができます。エージェント監視は、通信チャネルとして UNIX ソケット（Linux、MacOS）または TCP ソケット（Windows）のいずれかを使用して、gRPC API を介してカスタム・エージェントと通信します。API は、一致インストルメンテーション・イベントを呼び出します（インストルメンテーションがインストルメンテーション用の API メソッドを呼び出すなど）。これは Hashicorp の go プラグインを使用して実装されます。フレームワークには、Golang を使用して gRPC 通信がすでに実装されているソリューションが含まれています。エージェント・インターフェイスを実装するだけで、カスタム機能をフレームワークに追加できます。

用意されているエージェント

Agent Instrumentation Framework には、いくつかの内蔵エージェントが用意されています。ここで、その使用方法と機能について説明します。詳細については、選択した Defensics プロトコル・スイートを参照してください。

LogTailerAgent

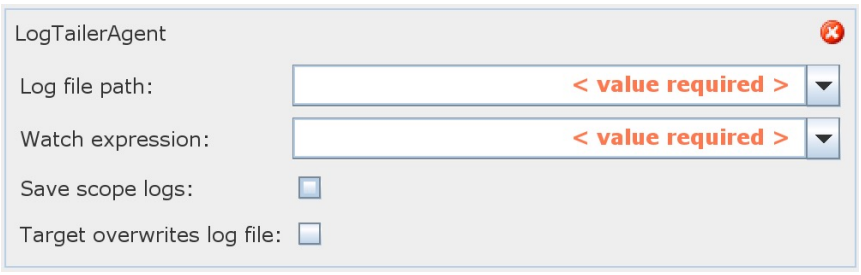


図 5：LogTailerAgent の構成オプション

LogTailerAgent は、作成後のログ・ファイルを監視します。Golang の正規表現を使用して、ターゲット・システムのエラーを検出することができます。ログ・ファイルに書き込まれた行がこの正規表現に一致する場合、その行からインストルメンテーションが失敗します。

FileMonitorAgent

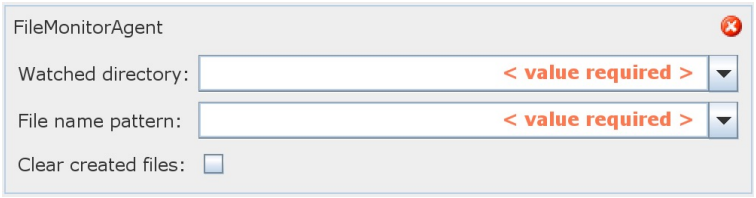


図 6：FileMonitorAgent の構成オプション

FileMonitorAgent は、ディレクトリ内でのファイルの作成を検索します。Golang の正規表現を使用して、ファイル名を照合することができます。インストルメンテーションが失敗するには、ターゲット・ファイル名がこの操作の正規表現に一致する必要があります。たとえば、この設定が値 AB に割り当てられている場合は、[watched directory] 設定で指定したディレクトリ内に ABB という名前のファイルを作成すると、インストルメンテーションの失敗がトリガーされます。同じディレクトリ内に ACC という名前のファイルを作成しても、インストルメンテーションの失敗はトリガーされません。チェックボックスを選択すると、ファイルが削除され、今後参照するためにエージェントの作業ディレクトリに保存されます。この設定の目的は、今後エラーが発生したときに既存のファイルによってファイルを作成できなくなった場合にテストを続行できるようにすることです。

ProcessMonitorAgent

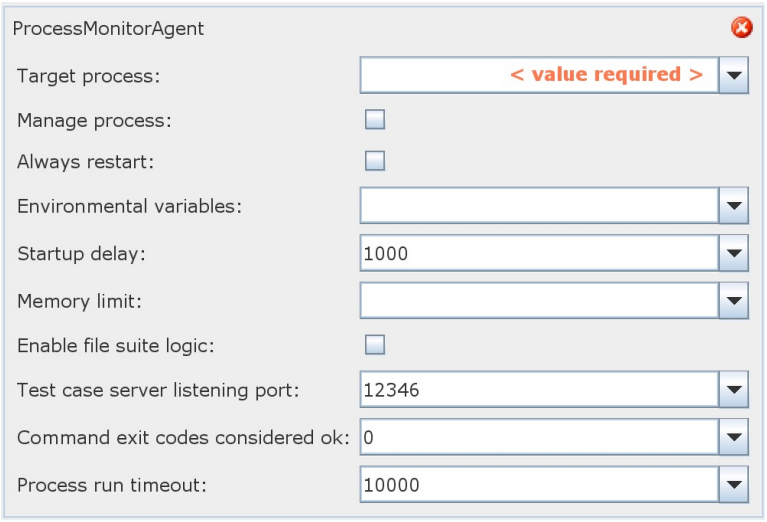


図 7：ProcessMonitorAgent の構成オプション

ProcessMonitorAgent は、プロセスの状態を監視します。このエージェントは、ターゲット・プロセスを開始するか、すでに実行されているプロセスを監視することができます。このインストルメンテーション手法では、ターゲット・プロセスがダウンしたり、再開されたり、ゾンビ・プロセスに変化した場合に、失敗と判定されます。デーモンを監視するプロセスは、多くの場合、クラッシュしたプロセスを数ミリ秒以内に再開させます。そのため通常のインストルメンテーションの場合では検知されません。このエージェントは、そのようなイベントのほか、クライアント側テストの場合の SUT プロセス論理などを捕らえます。

既存のプロセスの場合は、指定した値がターゲット・システム上で実行されているプロセスの名前と照合されます。部分一致もサポートされていますが、コマンドのコマンド・ライン全体を指定する必要はありません。プロセスは、管理することができます。つまり、インストルメンテーションが失敗した場合や、[Always restart] オプションが使用されている場合の各テスト・ケース後に、エージェントによってターゲット・プロセスが再開されます。さらに、事前定義されたメモリしきい値を超えていないかプロセスを監視することができます。

このエージェントには、Defensics ファイル形式スイートのインジェクションやインストルメンテーションの高度なワークフロー・オプションがほかにいくつかあります。ファジング対象ネットワーク・トラフィックの代わりにファジング対象ファイルを生成する Defensics スイートがあり、それらを解析するにはコマンドの実行が必要です。ProcessMonitorAgent は、生成されたファジング対象コンテンツを TCP サーバーから自動的に取得したり、ファジング対象コンテンツがファイル・システムに書き込まれたときに直接コマンドを実行したりすることができます。

SanitizerProcessMonitorAgent

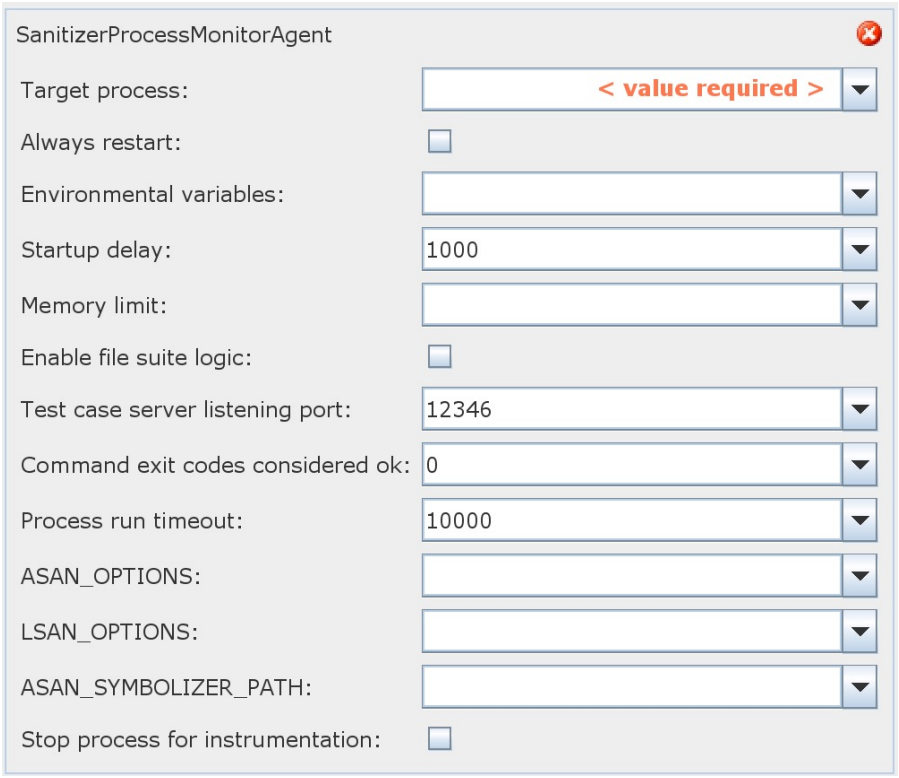


図 8：SanitizerProcessMonitorAgent の構成オプション

SanitizerProcessMonitorAgent は、Google の ASAN フレームワークを使用して、ソフトウェアにおけるメモリのアドレス指定能力の問題やメモリ・リークを見つけます。これにより、以下のような問題を見つけることができます。

- Use After Free（ダングリング・ポインタの間接参照）
- ヒープ・バッファ・オーバーフロー
- スタック・バッファ・オーバーフロー
- グローバル・バッファ・オーバーフロー
- Use After Return
- Use After Scope
- 初期化順序バグ
- メモリ・リーク

このエージェントを正しく運用するには、Linux と Mac でのみ利用可能な追加のコンパイラ・フラグを使用してターゲット・ソフトウェアをコンパイルする必要があります。ファイル形式やメモリしきい値のサポートなど、ProcessMonitorAgent で利用可能なすべてのオプションに加えて、微調整された構成に対して ASAN オプションと LSAN オプションを設定できます。一部のシナリオでは、ターゲット・プロセスを完全に終了する必要があります。これは [Stop process for instrumentation] 機能を介して行うことができます。

このエージェントでは、ターゲット・アプリケーションの非常に的確で詳細なインストルメンテーションが可能ですが、テスト・ケースの実行中にオーバーヘッドが加えられます。多数のテスト・ケースの実行時に、並列テスト設定で、他のインストルメンテーションを使用せずに、より高度なエージェントをデプロイすることをお勧めします。

KubernetesAgent

図 9：KubernetesAgent の構成オプション

KubernetesAgent は、Defensics が Kubernetes クラスタおよび監視ポッドに接続できるようにします。このエージェントは、外部 Kubernetes API と通信するため、SUT 自体で実行される必要がありません。このエージェントは、エージェント監視と同じシステム上で実行され、そこからクエリーを実行します。構成が kubeconfig ファイルから自動的に読み込まれ、その後のフィルタリングがコンテキスト名、名前空間、ラベル・セレクタを介して実行されます。

指定した名前空間は、リソース変更がないか監視されます。その後の監視制限は、ラベル・セレクタ構成を使用して設定できます。複数のラベルをカンマ区切りのリストとして指定できますが、指定しなかった場合は、名前空間内でサポートされているすべてのリソースが監視されます。

KubernetesAgent は、ダウンして再起動しているポッドを検出できます。これにより、テスト・ケースの失敗がトリガーされます。Kubernetes のデプロイメントでは、多くの場合、冗長性に注目されます。そうでなければ、そのようなイベントは気付かれないでしょう。Kubernetes のログは、修正やバックトレースを行いやすくするために、Defensics ログ・ファイルにも含まれています。

実装とテスト結果

インフォテインメント・システムにおける未知の脆弱性を見つける際の Agent Instrumentation Framework の効果を実証するために、ファジングの実行中にさまざまなエージェントを備えたフレームワークを使用しました。ブラック・ダックの Defensics ファザーが使用されたことにより、ユーザーは、プロトコル固有の RFC 準拠シーケンス、SNMP、syslog、機能プロトコル・チェックなど、デフォルトのインストルメンテーション機能を簡単に拡張できます。

テスト対象は、OEM やアフターマーケット・ソリューションを含めた、さまざまなベンダーのインフォテインメント・システムでした。ターゲットの 1 つとして、エミュレートされた環境で実行されている抽出されたオペレーティング・システムがありました。

一般的なインフォテインメント・システムではさまざまなプロトコルがサポートされており、機能性と接続性を追加することによって攻撃対象領域が拡張されます。完全な攻撃対象領域解析についてはこのホワイトペーパーでは触れませんが、注目したインターフェイスとプロトコルはほとんどが外部にアクセスできます。

Bluetooth のファジング

Defensics には、隣接する Bluetooth デバイスの位置を特定してその Bluetooth デバイスとペアリングし、テスト・スイートを設定するためのスキャン・オプションが含まれています。スキャンが Bluetooth 対応デバイスの位置を特定するには、そのデバイスが検出可能である必要があり、ほとんどの場合は設定を手動で有効にする必要があります。

スキャンにより、周辺で検出された全デバイスのリストが返されます。デバイスで有効になっているサポート対象サービスおよびプロファイルに関する詳細情報は、目的のデバイスのサービス検出機能を使用して取得できます。その後、この情報は Defensics に自動的にインポートされ、プロトコルで使用されるシーケンスの相互運用性テストの後にテスト・ケースが生成されます。

Bluetooth には、多くの場合、さまざまなアプリケーションのペイロードが含まれており、インフォテインメント・システム上で低レベルの権限で実行されます。そのため、そのプロセスとデーモンは、ブラック・ダックのエージェントが注目する候補に適していました。Bluetooth は、ドアロックの解除や車両情報へのアクセスなど、車内および車両周辺の重要な操作にも使用されます。

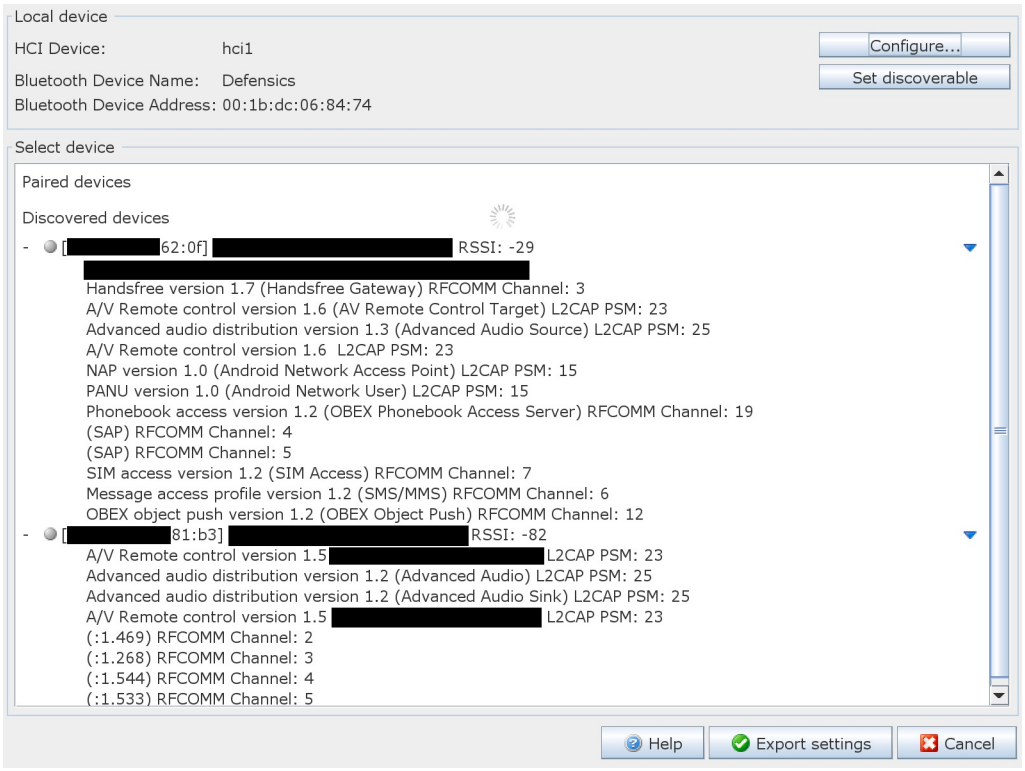


図 10：Defensics の Bluetooth デバイス・スキャンのダイアログ

テスト結果

インフォテインメント・システムに対するテストにおいて、重要な脆弱性が見つかりました。バッファ・オーバーフローの異常を含む 1 つのフレームによって、主要な Bluetooth カーネル・モジュールがクラッシュしました。これは、ProcessMonitorAgent を使用したテスト・ケースの実行中に、bluetoothd デーモン・プロセスを監視することで見つけることができます。bluetoothd を迅速に再起動するデーモン・ウォッチドッグがデバイスにあり、この特別なインストルメンテーションがなかったら、このクラッシュを検知することはできなかったでしょう。

懸念されるのは、オーバーフロー・ケースになる異常と、コア・カーネル・モジュールをクラッシュさせる異常の組み合わせです。これにより、root 権限で実行されているターゲット・プロセスがさらに悪用される可能性があるからです。エージェントにより、クラッシュが捕らえられ、さらなる調査のために stderr 出力が Defensics ホスト上のローカル・ファイルにコピーされました。

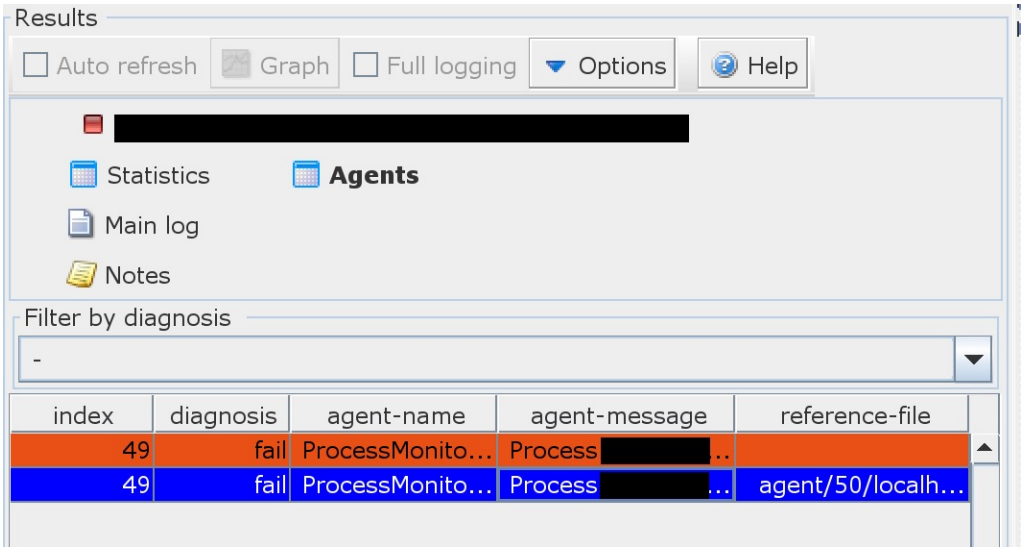


図 11：エージェントの失敗を示す Defensics のインストルメンテーション

ワイヤレスのファジング

802.11 のプロトコル・ファミリーでは、Defensics FuzzBox WLAN のスキャン機能を使用できます。これにより、必要なパラメータが、選択したターゲット・デバイスに基づいて、対応する設定フィールドにコピーされます。

インフォテインメント・システムでは、多くの場合、搭乗者が 4G/5G 接続を使用できるように、車内のワイヤレス・ネットワークを構築することができます。802.11 の実装で新たな脆弱性が見つかると、それは魅力的な標的になります。ワイヤレス・クラウド内のすべてのデバイスにアクセスできるようになり、Bluetooth よりも長距離からアクセスされる可能性があるからです。

テスト結果

テスト・セッション中に、いくつかのカーネル・モジュールをクラッシュさせるバッファ・オーバーフローの異常を含むフレームが 1 つ見つかりました。興味深いことに、これは非認証フレームであるため、理論的には誰かが送信した可能性があります。懸念されるのは、オーバーフロー・ケースになる異常と、コア・カーネル・モジュールをクラッシュさせる異常の組み合わせです。これにより、ターゲット・システムがさらに悪用される可能性があるからです。

影響を受けるモジュールはカーネル・ウォッチドッグによってすぐに再起動され、明らかな切断がなかったため、Defensics はこの脆弱性を検出できなかったでしょう。カーネルでコア・ダンプの機能を有効にしていた場合は FileMonitorAgent を使用してコア・ダンプが作成されていることを検知、またはカーネル・クラッシュの影響を受けるプロセスを監視する ProcessMonitorAgent を使用して検知することができたでしょう。代わりに、LogTailerAgent を使用し、「stack」、「crash」といったキーワードやいくつかのカーネル・モジュール名を syslog の後に付けることでこの問題を見つけました。該当する syslogs 出力を図 12 に示します。

```
[  +0.000032] ------------[ cut here ]------------
[  +0.000019] WARNING: CPU: 3 PID: 912 at drivers/net/wireless/[redacted]
[  +0.000002] Modules linked in: loop(O) [redacted]
[  +0.000083] CPU: 3 PID: 912 Comm: [redacted] Tainted: G     U  W  O [redacted]
[  +0.008363] task: edfbab40 task.stack: ecf66000
[  +0.005063] EIP: iwl_mvm_tx_mpdu+0x1a7/0x3d7 [iwlmvm]
[  +0.000003] EFLAGS: 00010286 CPU: 3
[  +0.000002] EAX: 0000001f EBX: ee75cde4 ECX: f4670344 EDX: f466ab4c
[  +0.000002] ESI: 00000002 EDI: 000001a0 EBP: ecf67bdc ESP: ecf67ba0
[  +0.000003]  DS: 007b ES: 007b FS: 00d8 GS: 0000 SS: 0068
[  +0.000002] CR0: 80050033 CR2: a63de000 CR3: 2bcd8ec0 CR4: 001006f0
[  +0.000002] Call Trace:
[  +0.002741]  iwl_mvm_tx_skb+0x5b/0x139 [iwlmvm]
[  +0.005071]  iwl_mvm_mac_tx+0x9c/0x144 [iwlmvm]
[  +0.005068]  ? iwl_mvm_stop_ap_ibss+0x12e/0x12e [iwlmvm]
[  +0.005952]  ieee80211_tx_frags+0x17b/0x192 [mac80211]
...
```

図 12：802.11 カーネル・モジュールのクラッシュ中の syslog 出力

MQTT のファジング

MQTT などの軽量プロトコルは、単純および軽量という性質により、従来のプロトコルと比較していくつかの利点があります。MQTT は、クラウドで組み込みデバイスにメッセージを公開および受信させる、単純なプロトコルです。HTTP などのプロトコルと比較してパケットのオーバーヘッドが最小限であるため、非常に効率的で、低電力環境に適しています。MQTT は、自動車コンポーネントでも使用されます。

MQTT ブローカーのソースにアクセスできたため、SanitizerProcessMonitorAgent を使用して、メモリのアドレス指定能力の問題やメモリ・リークがないかテストすることができました。そのために、追加のコンパイル・フラグを使用してコードを再コンパイルして、実行中にエージェントが使用する環境変数によって制御される Google の ASAN インストルメンテーションを有効にしました。

テスト結果

SanitizerProcessMonitorAgent を使用して、一般的な MQTT ブローカーである Mosquitto でメモリ・リークが見つかりました。その後、この脆弱性は報告、修正されました。デフォルトで Defensics によって生成されるすべてのテスト・ケースの実行時に、インストルメンテーションを追加しないでこの問題を検出することはできません。RFC で定義されている正しい MQTT Connect メッセージの例を図 13 に示します。

図 13：Defensics での正しい MQTT Connect メッセージ

エージェントでいくつかの失敗テスト・ケースが報告された後、実行が継続されました。すべてのテスト・ケースで同様の異常が使用されました。図 14 に示すように、SUT への送信前にパケットのバイトが削除された、アンダーフローの異常です。

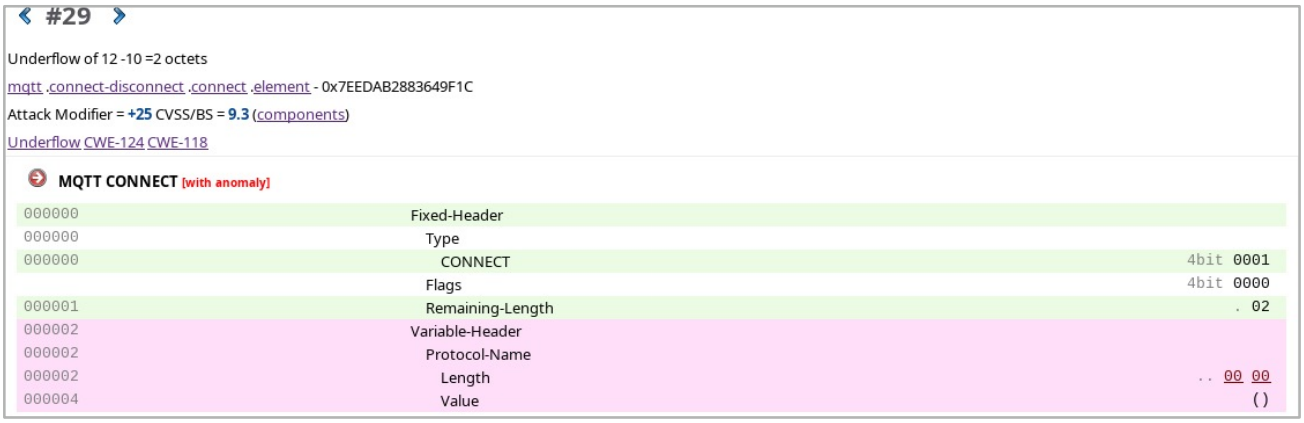


図 14：Defensics での MQTT の異常の例

各テスト・ケースで、アンダーフローのバイト数が以前よりも多くなりました。図 15 に示すように、使用したエージェントに応じて、合計 5 つのテスト・ケースが失敗しました。

test-group	index	status	input-octets	output-oct...	diagnosis	time	instrument...
mqtt.conn...	25	MQTT ...	4	10485764	pass	2.060	1
mqtt.conn...	26			10485763	pass	1.179	1
mqtt.conn...	27			2	pass	0.861	1
mqtt.conn...	28			3	pass	0.809	1
mqtt.conn...	29			4	fail	0.995	2
mqtt.conn...	30			5	fail	0.938	2
mqtt.conn...	31			6	fail	0.862	2
mqtt.conn...	32			7	fail	0.867	2
mqtt.conn...	33			8	fail	0.871	2
mqtt.conn...	34			9	pass	0.851	1
mqtt.conn...	35			10	pass	0.170	1

図 15：Defensics で失敗のマークが付けられた 5 つのテスト・ケース

詳細なログには、バイトごとのアンダーフローの増加のほか、アンダーフローによって増加するリークしたバイト数も示されています。図 16 に示すように、エージェントは Defensics にリークのトレースも提供します。

```
21:34:37 TEST CASE #29
21:34:37 mqtt.connect-disconnect.connect.element: Underflow of 12 -10 =2 octets
21:34:37 tcp 45264 --> localhost:1883 4 MQTT CONNECT ANOMALY!
21:34:37 Receiving connack over tcp failed: expected (0b0010) but got ()
21:34:37 Instrumenting (1. round)...
21:34:37 /usr/bin/python2 /home/p0c/synopsys/aif/client.py      --config /home/p0c/synopsys/aif/configs/mqtt-asan.json instrumentation
21:34:37 Instrumentation verdict: FAIL
21:34:37 FAIL Agent: memory_mqtt Info: Agent memory_mqtt says Memory leak found in /home/p0c/mosquitto/src/mosquitto:
21:34:37
21:34:37 =================================================================
21:34:37 ==20==ERROR: LeakSanitizer: detected memory leaks
21:34:37
21:34:37 Direct leak of 1 byte(s) in 1 object(s) allocated from:
21:34:37     #0 0x7f6af581ed99 in __interceptor_malloc /build/gcc/src/gcc/libsanitizer/asan/asan_malloc_linux.cc:86
21:34:37     #1 0x56218adca9e3 in _mosquitto_malloc (/home/p0c/mosquitto/src/mosquitto+0x3d9e3)
21:34:37     #2 0x56218ade0802 in _mosquitto_read_string (/home/p0c/mosquitto/src/mosquitto+0x53802)
21:34:37     #3 0x56218ade5d85 in mqtt3_handle_connect (/home/p0c/mosquitto/src/mosquitto+0x58d85)
21:34:37     #4 0x56218ade2e77 in mqtt3_packet_handle (/home/p0c/mosquitto/src/mosquitto+0x55e77)
21:34:37     #5 0x56218ade2b61 in _mosquitto_packet_read (/home/p0c/mosquitto/src/mosquitto+0x55b61)
21:34:37     #6 0x56218adca6b7 in loop_handle_reads_writes (/home/p0c/mosquitto/src/mosquitto+0x3d6b7)
21:34:37     #7 0x56218adc891c in mosquitto_main_loop (/home/p0c/mosquitto/src/mosquitto+0x3b91c)
21:34:37     #8 0x56218ada185c in main (/home/p0c/mosquitto/src/mosquitto+0x1485c)
21:34:37     #9 0x7f6af430606a in __libc_start_main (/usr/lib/libc.so.6+0x2306a)
21:34:37
21:34:37 SUMMARY: AddressSanitizer: 1 byte(s) leaked in 1 allocation(s).
21:34:37
```

図 16：エージェントによって提供された詳細が追加されたメモリ・リーク

ファイル形式のファジング

インフォテインメント・システムの一般的な機能は、豊富なメディア・コンテンツを再生する機能です。単純なシステムで再生されるのは音声ファイル形式のみですが、大きなディスプレイを備えた高価なシステムでは動画や画像コンテンツも再生されます。これらのデバイス上のファイル形式パーサーは、受信した入力に埋め込まれたエクスプロイトに対して脆弱です。

Defensics には、一般的な各種ファイル形式の完全な仕様に基づいてファジング対象バージョンを生成できる、ファイル形式ファザーがいくつかあります。ファイル形式ファザーをディスクに書き込んだり、他の論理を使用してこれらをソフトウェア入力に送信して判定を行ったりすることができます。

テスト結果

音声と動画の再生機能をテストするために、Defensics の内蔵ファイル形式論理を使用して、ファジング対象ファイルを各種インフォテインメント・システムに送信して自動的に再生し、エージェントを使用して判定を行いました。ここでエージェントの組み合わせを使用してさまざまなソースを実装しましたが、設定を最小限に保つために ProcessMonitorAgent を選択しました。

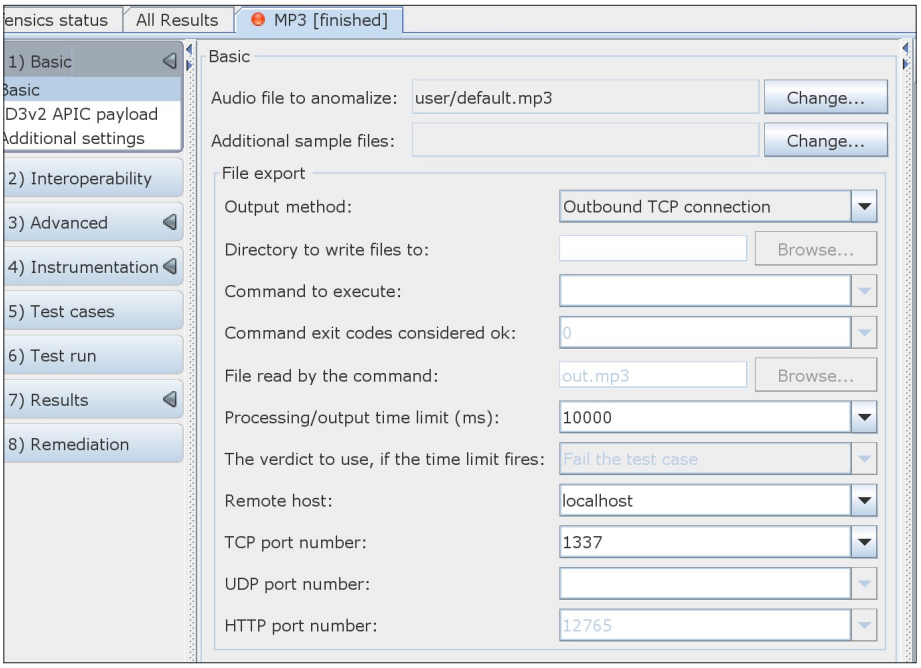


図 17：Defensics の MP3 ファイル形式スイートの構成

ファイル形式スイートの構成は簡単明瞭です。生成された（ファジング対象の）各ファイルは、[Outbound TCP connection] オプションでエクスポートされます。つまり、指定した IP アドレスとポートの組み合わせに TCP によって送信されます。ここでのポートは、SUT 上で実行中の ProcessMonitorAgent で設定されたポートと同じです。

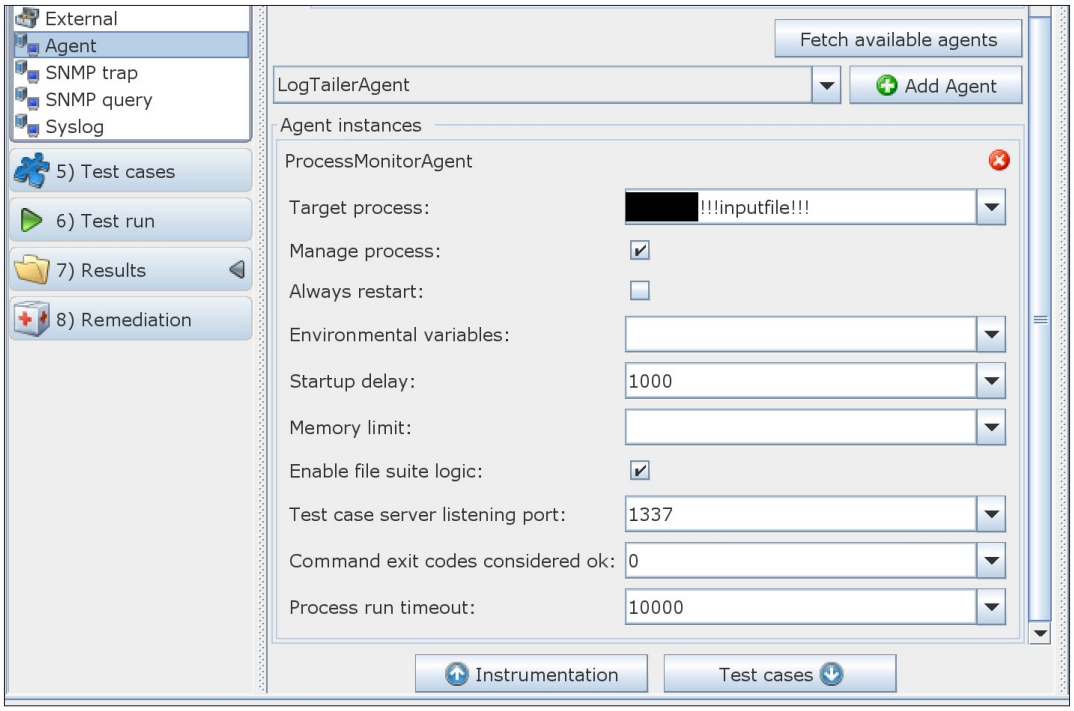


図 18：ProcessMonitorAgent の構成

テスト・ケースが実行されると、ファイルは SUT 上で実行中のエージェントに送信されます。エージェントの構成でファイル・スイート論理を有効にしているからです。また、ターゲット・プロセスにファジング対象ファイルを解析させるために必要なコマンドを指定しました。[Target process] オプションで、ターゲット・プロセスの引数として使用されるファイル名のプレースホルダとして「!!!inputfile!!!」を使用しました。これは実行中にエージェントの論理によって処理されます。インストルメンテーションは、エージェントのデフォルト構成によって自動的に実行されます。

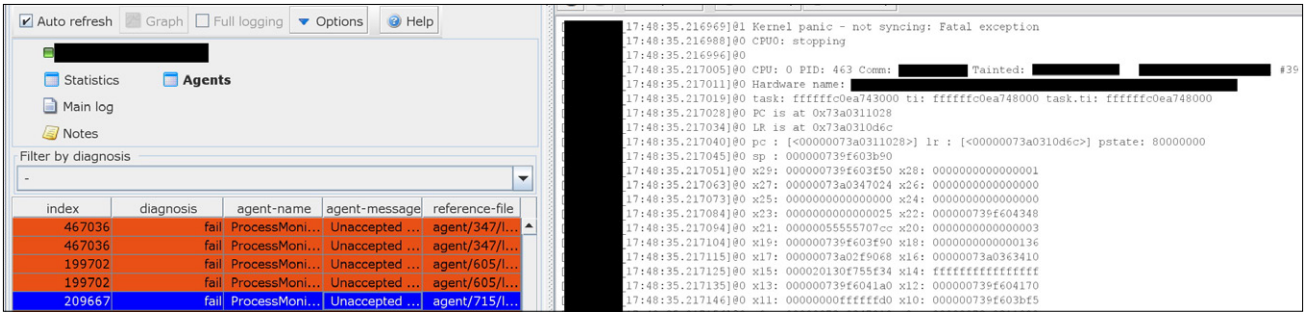


図 19：ProcessMonitorAgent の報告された失敗テスト・ケース

テスト中に、いくつかのクラッシュを観察しました。テストしたインフォテインメント・システムへの影響として、プロセスのクラッシュ、カーネル・パニック、ある事例では完全な再起動がありました。これらの深刻な問題は、音声、動画、および画像ファイル形式スイートで利用可能なテスト・ケースの小規模なサブセットの実行中にのみ、見つかりました。

最後に

このホワイトペーパーでは、Agent Instrumentation Framework を紹介し、それを使用してどのように HPC のファジング・テストを改善できるかを説明しました。特に、インフォテインメントと接続装置に注目しました。それらのターゲット・システムを適切に実装して、効率的で正確なファジング・テストを可能にする方法について説明しました。SUT にデプロイされた 1 つまたは複数のエージェントを使用して、SUT 上のテスト・ケースによって例外が発生したかどうかを判定するための追加情報を収集することができます。この情報は、ファジング・テスト・ツールにも提供され、ログ・ファイルに保存されます。それによって開発者は検出された問題の根本原因を特定し、問題を効率的に修正することができます。提案したフレームワークの効果を示すために、この手法に基づいてテスト・ベンチを構築し、いくつかの SUT のファジング・テストを実行しました。結果を提示し、エージェントのインストルメンテーションがなかったら SUT 上の問題が検出されなかった可能性がある例をいくつか挙げました。

Agent Instrumentation Framework は、HPC に最適です。Agent Instrumentation Framework は、通常、Linux や Android など、より多くの機能を備えたオペレーティング・システムを使用し、SUT 上でエージェントを実行できるようにします。ブラック・ダックは、コネクテッド・カーや自動運転における成長により、自動車業界の大量のソフトウェア開発が推進され、自動車開発プロセスのサイバー・セキュリティに対する意識の高まりと相まって、自動化されたファジング・テストがこのようなタイプのシステムに必須のステップになると考えています。ブラック・ダックが提案するフレームワークは、Linux や Android などの機能が豊富なオペレーティング・システムを実行する SUT に対して、自動化されたファジング・テストをサポートすることができます。

参考文献

[1] D.K. Oka, “Building Secure Cars: Assuring the Automotive Software Development,” Wiley, 2021.

[2] S. Bayer, T. Enderle, D. K. Oka, and M. Wolf, “Security Crash Test—Practical Security Evaluations of Automotive Onboard IT Components,” in Automotive—Safety & Security 2015, 2015.

[3] D. K. Oka, A. Yvard, S. Bayer, and T. Kreuzinger, “Enabling Cyber Security Testing of Automotive ECUs by Adding Monitoring Capabilities,” in escar Europe, 2016.

[4] D. K. Oka, T. Fujikura, and R. Kurachi, “Shift Left: Fuzzing Earlier in the Automotive,” in escar Europe, 2018.

[5] Automotive Grade Linux, Automotive Grade Linux, accessed May 6, 2018.

[6] Automotive Grade Linux, Automotive Grade Linux Hits the Road Globally with Toyota; Amazon Alexa Joins AGL to Support Voice Recognition, accessed May 7, 2018.

[7] GENIVI, accessed May 6, 2018; Open Automotive Alliance, Introducing the Open Automotive Alliance, accessed May 6, 2018.

[8] BlackDuck, Heartbleed Bug, accessed May 7, 2020.

ブラック・ダックについて

ブラック・ダックは、業界で最も包括的かつ強力で信頼できるアプリケーション・セキュリティ・ソリューション・ポートフォリオを提供します。ブラック・ダックには、世界中の組織がソフトウェアを迅速に保護し、開発環境にセキュリティを効率的に統合し、新しいテクノロジーで安全に革新できるよう支援してきた比類なき実績があります。ソフトウェア・セキュリティのリーダー、専門家、イノベーターとして認められているブラック・ダックは、ソフトウェアの信頼を築くために必要な要素をすべて備えています。

詳しくは www.blackduck.com/jp をご覧ください。

ブラック・ダック・ソフトウェア合同会社

www.blackduck.com/jp