Vol.2021-OS-152 No.4
2021/5/27

Sprofiler: ワークロードにフィットしたコンテナのSeccompルール生成システムの開発と評価

飯國 隆志[1]　最所 圭三[1]

概要：近年，コンテナ型仮想環境をプロダクション環境で使用する例が増加している．コンテナ型仮想化はパブリッククラウドのようなマルチテナントな環境で運用するにはセキュリティ問題を抱えている．コンテナランタイムでは，コンテナ-ホスト間でカーネルを共有する特性上，コンテナ内からコンテナホストへの権限昇格攻撃を可能にする脆弱性が生まれやすい．脆弱性がなくとも設定不備によって権限昇格が可能になってしまうこともある．それらの対策として Seccomp によるコンテナ内で発行するシステムコールを制限する方法が挙げられる．しかし，システム管理者がアプリケーションの発行するシステムコールを把握するのは難しい．そのため本研究では，アプリケーションの実行ファイルを解析した結果と実際のコンテナ内で発行されたシステムコールを記録した結果を合わせて，ワークロードに合ったシステムコールの制限ルールをホワイトリストで出力するアプリケーション Sprofiler を提案する．本稿では，Sprofiler の設計，実装および Sprofiler で生成した Seccomp ルールの評価について述べる．

キーワード：コンテナ，Linux, システムコール，Seccomp, 権限昇格

1. はじめに

コンテナ型仮想化は軽量な仮想化手法として，近年広く使われるようになった．Kubernetes[1] などのコンテナオーケストレーションソフトウェアの発展が著しく，国内でもコンテナ型仮想化をプロダクション環境で使用している企業は増加傾向にある [2]．金融関係のシステムで導入された例もあり，今後も需要は増加していくことが予想される．しかし，コンテナ型仮想化はコンテナ-ホスト間で OS カーネルを共有するという性質上，以下に示すようなセキュリティ面の課題を抱えている．

- VM などに比べ，コンテナ間の隔離性が低い
- ホストの OS カーネルに対する攻撃がホスト内の全コンテナに影響する

IDC Japan の調査結果では，コンテナや Kubernetes の導入の課題として「セキュリティ対策」(30.2%) が「障害・問題発生時の対応策」(32.5%) に次いで 2 位だった [2]．コンテナのプロダクションの導入にあたって，セキュリティ対策の導入は必須と言える．

本研究では，コンテナのセキュリティ問題の中でも権限昇格攻撃の対策としてコンテナ内のシステムコールを制限することに着目する．Seccomp を使用してコンテナの発行するシステムコールを制限し，Attack Surface を小さくするにはコンテナ上で動作するアプリケーションごとに発行するシステムコールを把握する必要がある．そのため本研究では，アプリケーションの実行ファイルを解析した結果と実際のコンテナ内で発行されたシステムコールを記録した結果を合わせて，ワークロードに合ったシステムコールの制限ルールをホワイトリストで出力するアプリケーション Sprofiler を提案する．本稿では，Sprofiler の設計，実装および Sprofiler で生成した Seccomp ルールの評価について述べる．

[1] 香川大学
Kagawa Uniersity

2. コンテナへの権限昇格攻撃

コンテナのセキュリティ問題の中でも，特にコンテナから不正にコンテナホストの権限を獲得する権限昇格攻撃への懸念が大きい．権限昇格攻撃はコンテナイメージの改ざん，遠隔コード実行，コンテナエンジンの API の不正利用などの脆弱性を利用することで行われる．近年広く使用されているコンテナ型仮想環境 Docker などでもいくつか脆弱性が報告されている．2014 年には，第三者により，細工されたコンテナイメージやビルドを介し，管理者権限で任意のコードを実行される脆弱性が報告された (CVE-2014-9357)[3]．脆弱性は Docker のアーカイブ抽出処理における chroot システムコールに関する処理に不備

ⓒ 2021 Information Processing Society of Japan

によるものである．2019 年では，Docker の内部で使用されているコンテナランタイム runc に，コンテナランタイム自身のバイナリがコンテナ内部から上書きされ，ホストの root 権限で任意のコマンドを実行可能になる脆弱性が発見された (CVE-2019-5736)[4]．これらの脆弱性の深刻度を示す CVSS スコアは，「緊急」,「重要」などのに分類されるほどの高いスコアであることが多い．こういった攻撃に対応するため，Linux カーネルの機能 (2.1 節) や，セキュアなコンテナランタイム (2.2 節) を使用してホスト-コンテナ間隔離を強化しようという試みがなされている．

2.1 Linux カーネルによる隔離機能

本節では，Linux カーネルに備わっているセキュリティ機構について述べる．

Capability は Linux の権限分離のための機能である．伝統的な UNIX の実装では，プロセスは特権プロセスと非特権プロセスに分類される．しかし，それでは一部の特権機能を必要とするプロセスに不要な特権を与えることとなる．Capability を用いると，特権をグループ (Capability) に分割し，それぞれ独立して有効，無効に設定できる．

Seccomp はシステムコールの発行を制限する機能である．ルールベースでシステムコールの発行にフックし，エラーを返したり，プロセスを kill するなどのアクションの設定が可能である．Docker では Seccomp は標準で有効化され，デフォルトプロファイルは与えられているものの，Capability が必要となるシステムコールを定義している程度である．そのため，Capability さえあれば，大半のシステムコールが実行可能となっている．Docker などで用いられる runc などの OCI Runtime Specification[5] に則ったコンテランタイムへ与える Seccomp のプロファイルは，アプリケーションが発行するものだけでなく，コンテランタイムが発行するシステムコールなどを含むため，単純なプログラムでも数十のシステムコールを許可しなければならない．コンテナランタイムの発行するシステムコールも実装によって異なる．

AppArmor や SELinux は Linux Security Modules の一種であり，強制アクセス制御機能 (MAC) を提供する．これにより，コンテナ内からコンテナ外のファイルへのアクセスを制限可能である．Docker は AppArmor の標準のルールが用意されている．

これらの手法の問題点は，コンテナが正常に動作し，コンテナ内に与える権限を最低限にするセキュリティポリシーを作成することが困難であることだ．管理者が Kubernetes などで動作するワークロードすべての権限やシステムコールなどを把握し，手動で Seccomp や AppArmor などのポリシーを作成するには，Linux カーネルへの知識や，多くのテスト工数が必要になる．

2.2 コンテナランタイムによるセキュリティ機構

コンテナは OS カーネルをコンテナホストと共有することのリスクに着目し，独自のセキュリティ機構を持つコンテナランタイムも存在する．

Google 社の開発する gVisor[6] はユーザ空間で動作する Linux 互換の OS カーネルの上でコンテナを実行するコンテナランタイムである．gVisor のコンテナで発行されたシステムコールは KVM または ptrace によってフックされ，検査し，危険なシステムコールは Seccomp によってフィルタされる．gVisor は Google Cloud Platform 内のサービスで実際に使われており，Cloud Run や GKE Sandbox など，マルチテナントな環境のセキュリティを強固にする必要があるワークロードのためのクラウドサービスに用いられている．gVisor はシステムコールの割り込みへのオーバヘッドが大きいため，システムコールの発行を多用するワークロードには向かないとされている．また，ユーザ空間で実行するための Linux カーネルの互換性が特定のカーネルのバージョンに依存する他，実装されていないシステムコールもいくつか存在する．

Kata Containers[7] は，コンテナを microVM 上で実行するためのコンテナランタイムである．ハードウェア仮想化によって，ネットワーク，ディスク IO，メモリへの強力な分離を提供できる．欠点として，Kata Containers の動作要件にはベアメタル環境または Nested Virtualization が有効化された Hypervisor を必要とするため，運用するプラットフォームが限定されること，起動速度の低下やということが挙げられる．そのため，VM ベースの IaaS などでは実現が難しく，Nested Virtualization が有効な場合でもパフォーマンスの低下は免れない．

3. 関連研究

関連研究として, Linux やコンテナにおけるセキュリティポリシーを生成するという研究やシステムを例にあげる．

3.1 TOMOYO Linux

TOMOYO Linux[8] は，強制アクセス制御 (MAC) の実装の 1 つである．SELinux や AppArmor などの既存の MAC の実装と異なる点は，システムを解析し，ポリシーを自動生成できることにある．TOMOYO Linux 1 系では，Linux カーネルへのパッチ形式で実装されていたため，Linux カーネルをコンパイルし直す必要があるなどのデメリットがあった．TOMOYO Linux 2 系からは LSM(Linux Security Module) のインタフェースを利用し, 標準の Linux カーネルに統合された．

3.2 docker-slim

docker-slim はコンテナイメージのサイズの最小化と安全性を確保するツールである．コンテナ内の挙動を ptrace シ

ステムコールによってトレースし，AppArmor や Seccomp のプロファイルを作成できる．現在対応しているコンテナ型仮想環境は Docker のみである．

3.3 oci-seccomp-bpf-hook

oci-seccomp-bpf-hook は 動的にコンテナ内のシステムコールをトレースし，Seccomp のルールを生成するツールである．コンテナ内のシステムコールのトレースには eBPF(extended Berkeley Packet Filter) を使用しており，システムコールの発行時にフックし記録する．これにより，ptrace システムコールより高速かつ安全 (レジスタを書き換えられる心配もなく) システムコールをトレース可能である．

4. Seccomp ルール生成システム Sprofiler

本節では，Sprofiler の要件，設計および実装状況について述べる．

4.1 セキュリティポリシーの動的生成における問題

コンテナやプロセスの挙動からセキュリティポリシーを動的に生成する場合，使用するアプリケーション内のコマンドラインオプションや分岐，例外などの挙動をすべて網羅するテストを行う必要がある．それらすべてのパターンをテストするのはアプリケーションの実装を把握していても困難である．仮にカバレッジが 100%のソフトウェアの自動テスト中に ptrace などを使用してシステムコールをトレースしたとしても，Mock や Stub での挙動がプロダクションと異なるため不可能である．

4.2 Sprofiler の要件

本研究では，コンテナのセキュリティを向上させるための手法を考案する．2.1 節で述べた Linux カーネルの隔離機構は，アプリケーションで使用するための特権やシステムコールなどをカバーしたセキュリティポリシーをシステム管理者が手動で作成するのは工数が多くかかる．また，Docker などが標準で用意するセキュリティポリシーは多くのアプリケーションに対応するため，Attack Surface が大きい．2.2 節で述べたコンテナランタイムによるセキュリティ対策はプラットフォームへの制約が大きい．

以上のことから，本研究では，コンテナランタイムを変更することなく，容易にセキュリティポリシーを生成する手法として，以下の要件を満足する Seccomp ルール生成システム Sprofiler を提案する．

要件 1　既存の VM ベースの IaaS や Container as a Service(CaaS) 上で使用可能であること．コンテナランタイムは runc や crun などの標準で使用されているものを使う．

要件 2　システム管理者が自動で Seccomp のルールを生成できること．手動でテストする必要がなく，アプリケーションの挙動を知らなくともルールを生成できることが望ましい．

要件 3　許可するシステムコールを最小化して Attack Surface を小さくする．不要な外部コマンドなども実行させない．

4.3 Sprofiler の設計

Sprofiler の設計について述べる．Sprofiler は OCI Runtime Specification に則った Seccomp のルールを生成する機能のみを持つ．そのため，プロダクションのコンテナランタイムやシステム構成を変更することは無いため，要件 1 を満たす．

要件 2 を満たすには，Sprofiler が手動テスト無しで，アプリケーションの発行するシステムコールを網羅した Seccomp ルールを生成する必要がある．docker-slim や oci-seccomp-bpf-hook のようなシステムコールの動的解析のみでは，分岐を網羅するテストをする必要がある．Sprofiler は動的解析機能 (4.3.2 節) に加え，アプリケーションバイナリから発行されるシステムコールを解析する静的解析機能 (4.3.1 節) を用いて，アプリケーションの発行するシステムコールを網羅する．この 2 種類の手法で生成されたルールを合わせて 1 つのルールとして使用することで，要件 2 を満たす．

要件 3 を満たすには，ホワイトリスト形式で許可するシステムコールを定義するルールとする．Sprofiler でトレースできなかったシステムコールは実行を許可されないため，アプリケーションの発行するシステムコールが少ないほど，Attack Surface は小さくなる．

4.3.1 システムコールの静的解析機能

静的解析機能の構成を図 1 に示す．静的解析機能は，実行形式バイナリからシンボルテーブルを抽出し，使用するシステムコールのラッパー関数名を取得し，Seccomp ルールを生成する．これにより，分岐や例外などに関係なくアプリケーションが発行するシステムコールを取得可能である．静的解析機能でシステムコールを取得するための条件があり，それらを以下に列挙する．

- コンパイル言語であること．実行形式バイナリを作成できないインタプリタ言語などに用いることはできない．
- 静的リンクされたバイナリであること．動的リンクライブラリで使用された関数名を取得できない．
- シンボルテーブル付きでコンパイルすること．バイナリのサイズを小さくするためにシンボルテーブルは削られる傾向にある．
- 外部コマンドへ依存しないこと．外部コマンドの発行

ⓒ 2021 Information Processing Society of Japan

するシステムコールは呼び出し側のバイナリからはわからない.

コンテナランタイムで発行されるシステムコールや言語のランタイムライブラリで使用されるシステムコールは取得できないため，静的解析機能で生成されたルールのみではコンテナは起動できない (5.1 節). そのため，動的解析機能と併用することが必要である.

4.3.2 システムコールの動的解析機能

動的解析機能 (sprofiler-bpf) の構成を図 2 に示す. 動的解析機能はトレース対象のコンテナの起動にフックして，OS で発行されたシステムコールのうち，コンテナの Cgroup 内のプロセスで発行されたシステムコールのみを記録する. システムコールトレースには eBPF を用いており，カーネル内のシステムコール発行時のトレースポイント (sys_enter) にフックし，システムコールを記録する. 先に述べたように動的解析機能では docker-slim などと同様にアプリケーションの分岐や例外をカバーするにはかなりのテスト工数が必要となるが，アプリケーションが発行するシステムコールは静的解析機能がカバーするため，コンテナランタイムや言語のランタイムの発行するシステムコールのトレースのみを目的としている.

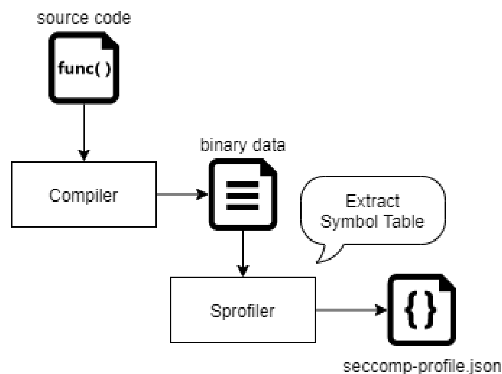

図 1 静的解析機能の構成

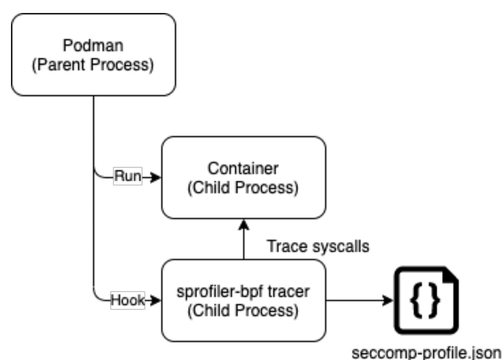

図 2 動的解析機能の構成

4.4 実装

現在 Go 言語のバイナリに対して静的解析機能を用いることができる (図 3). 今後は C 言語や Rust などに対応する予定である. 動的解析機能は Podman の hook として実装しており，コンテナの停止時に annotation で指定したパスに Seccomp ルールを出力するようになっている (図 4). 動的解析機能に Podman を用いる関係上，コンテナランタイムには crun を用いているが，Docker などで使用される runc にも対応予定である. 本システムによって生成された Seccomp ルール適用したコンテナで Dirty COW(CVE-2016-5195)[9] といった脆弱性を防ぐことが可能であることを確認した.

```
$ go build -o bin/sprofiler-demo-app ./...                      [1/1]
$ sprofiler run --bin bin/sprofiler-demo-app --out seccomp-profile.json
$ cat seccomp-profile.json
{"defaultAction":"SCMP_ACT_ERRNO","architectures":["SCMP_ARCH_X86_64"],
"syscalls":[{"names":["close","fcntl","mmap","munmap","readlinkat","wri
te"],"action":"SCMP_ACT_ALLOW"}]}$
```

図 3 静的解析機能の実行結果

```
# podman --hooks-dir $(pwd)/hooks \                                                      [3/3]
    run -d \
    --> > annotation "io.sprofiler.output_seccomp_profile_path=/tmp/seccomp-profile.json" \
    docke> r.io/guni1192/sprofiler-demo-app
5251ddccdc48824e0611f8756d6b4e67ee78687b6360f2e137cb0f1bdeb2450c
# cat /tmp/seccomp-profile.json
{"defaultAction":"SCMP_ACT_ERRNO","architectures":["SCMP_ARCH_X86_64"],"syscalls":[{"names"
:["access","arch_prctl","capset","chdir","clone","close","dup2","execve","exit_group","fchd
ir","fcntl","fstat","fstatfs","futex","getdents64","getegid","geteuid","getgid","getpid","g
ettid","getuid","lseek","mmap","mount","nanosleep","openat","pivot_root","prctl","read","re
adlinkat","rt_sigaction","rt_sigprocmask","rt_sigreturn","sched_getaffinity","seccomp","sel
ect","sethostname","setresgid","setresuid","setsid","sigaltstack","stat","statx","tgkill","
umask","umount2","uname","write"],"action":"SCMP_ACT_ALLOW"}]}#
```

図 4 動的解析機能の実行結果

5. 評価

本節では，Sprofiler の節で述べた要件 2 及び要件 3 に対する評価について述べる.

評価 1 トレース機能評価: 静的解析機能によって，動的解析機能でカバーできないシステムコールをカバーしきれているか (要件 2).

評価 2 Attack Surface の比較: Docker や Podman の標準のポリシーと比べて，いかにシステムコールによる Attack Surface を小さくできるか (要件 3).

評価実験環境を以下に示す.

- CPU: Intel Core i9-10900K
- Kernel: Linux 5.8.0-43-generic
- Distribution: Ubuntu 20.10
- OCI Runtime: crun 0.18.1-7931a-dirty
- Container Engine: podman 3.0.1

実験対象となるプログラムは，Hello World を標準出力に出力する Go 言語製のプログラム，Kubernetes クラスタ

上でコンテナとして用いられている CoreDNS[10], etcd[11] である．etcd は Kubernetes クラスタのメタデータを保存するための key-value Store として用いられる．CoreDNS は Kubernetes クラスタ内でアプリケーションのサービスディスカバリを行うために用いられる．今回用いるコンテナイメージを表 1 に示す．

表 1 実験対象のコンテナイメージ

実験対象	コンテナイメージ
Hello world	docker.io/guni1192/sprofiler-demo-app:0.1
CoreDNS	docker.io/coredns/coredns:1.8.3
etcd	quay.io/coreos/etcd:v3.2.32

システムコールの動的解析機能を用いて各コンテナ内で行うテスト内容について述べる．本研究では，テスト工数をかけずともシステムコールをトレースすることを要件とするため，アプリケーションの分岐などを網羅することはあえて行わない．今回の実験では，コンテナを起動後，アプリケーションへの API 呼び出しなどは行わず 5 秒後に停止する．

5.1 評価 1 トレース機能評価

静的解析機能と動的解析機能を使用し, 作成した Seccomp ルールを作成し，作成した Seccomp のルールを用いてコンテナの動作確認を行う．静的解析機能のみの場合，動的解析機能のみの場合，併用した場合に作成した 3 種類のルールを適用して実験した結果を比較する．これにより，動的解析機能でカバーしきれていないシステムコールが静的解析機能でカバーできているか評価する．

各コンテナの動作確認条件を述べる．HelloWorld プログラムについては，標準出力に「Hello world」が出力されれば正常終了とする．CoreDNS については外部，内部のドメインに対して，A レコード，CNAME レコード， MX レコードの正引きを行うことができれば正常動作とする．etcd についてはデータの挿入，データの取得ができれば正常動作とする．

実験結果を表 2 に示す．どのコンテナでも静的解析機能が生成したルールのみの場合，起動は成功しなかった．これはコンテナランタイムが使用するシステムコール (seccomp，prctl，mount など) がルールに含まれていなかったからである．動的解析機能で生成したルールのみの場合，起動には成功したものの，CoreDNS は名前解決に，etcd はデータの挿入に失敗し，動作確認の条件は満たせなかった．HelloWorld プログラムは起動から終了までに分岐もないため，動的解析機能でトレースしたシステムコールのみで動作は成功した．静的解析機能と動的解析機能を併用した結果，どのコンテナも動作確認の条件を満たすことができた．

図 5〜7 に動的解析機能と静的解析機能で許可したシステムコールの分布を示す．動的解析機能のみで生成したルールを用いて動作確認の条件を満たすことができた，HelloWorld プログラムは，静的解析機能のみで取得できたシステムコールは 1 個のみであった．一方，CoreDNS, etcd は，静的解析機能を用いなければ取得できなかったシステムコールが 20-30 程度存在した．

以上の結果から，静的解析機能によって，システムコールの動的解析で作成した Seccomp ルールより，テスト工数が少なく，動作確認できるルールが生成できたと言える．

表 2 動的解析機能と静的解析機能を用いたコンテナの動作確認

コンテナ	Seccomp ルール	動作確認
HelloWorld	dynamic + static	○
HelloWorld	dynamic	○
HelloWorld	static	×
CoreDNS	dynamic + static	○
CoreDNS	dynamic	×
CoreDNS	static	×
etcd	dynamic + static	○
etcd	dynamic	×
etcd	static	×

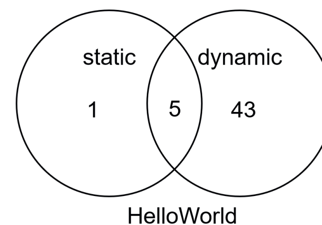

図 5 HelloWorld コンテナのシステムコール許可数

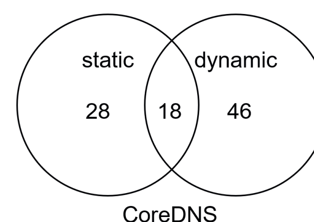

図 6 CoreDNS コンテナのシステムコール許可数

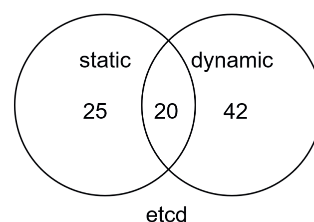

図 7 etcd コンテナのシステムコール許可数

5.2 評価 2 Attack Surface の比較

本実験では，動的解析機能と静的解析機能で生成したSeccomp ルールの Attack Surface を評価する．Docker や Podman などの既存のコンテナ型仮想環境で用いられているSeccomp ルールと，評価 1 で用いた動的解析機能と静的解析機能を併用したルールを比較する．

結果を表 3 に示す．Docker や Podman は 300 を超えるシステムコールを許可している．異なる CPU アーキテクチャのシステムコールも含むが，許可しているシステムコールは数十個程度である．Sprofiler を用いて生成したシステムコールは，今回対象としたアプリケーションでは 100 にも満たない．その大半が図 5〜7 から分かるとおり，動的解析機能のみで検出したものであることから，コンテナランタイムの実行に必要なシステムコールであることがわかる．最終的にどのコンテナも 200 以上のシステムコールを許しておらず，Attack Surface と減少させることに成功した．

表 3 Attack Surface の比較結果

コンテナ	Seccomp ルール	許可しているシステムコール数
-	Docker default	395
-	Podman default	344
HelloWorld	dynamic + static	49
CoreDNS	dynamic + static	92
etcd	dynamic + static	87

6. おわりに

本稿では，コンテナの権限昇格攻撃への対策として，Seccomp ルールの自動生成システムである Sprofiler の開発と評価について述べた．評価実験では，コンテナ内のアプリケーションの分岐などをすべて網羅するための手動テストをすることなく，コンテナが動作可能な Seccomp ルールを生成可能であることを確認した．そして，コンテナで実行を許可するシステムコール数を Docker や Podman などが標準で用意しているルールと比べ，200 以上のシステムコールの実行を制限し，Attack Surface を小さくすることを可能にした．

今後の機能開発の課題を以下に示す．

- Continus Integration(CI) への組み込みを可能にする．Sprofiler をバイナリで配布できるようライブラリを静的リンクすること，eBPF を使用可能な CI サービスの開発などを考えている．
- 静的解析機能の対応言語を増やし，様々なアプリケーションで動作検証を行う．
- デバッグ手段の検証，またはそのためのツール開発を行う．Sprofiler で生成したルールは，最小限のシステムコールしか許可していないため，システムの運用時にコンテナ内で対話シェルを用いたデバッグ作業などが難しい．今回の実験で使用した CoreDNS のコンテナイメージのように実行形式バイナリのみで構成されるコンテナイメージも存在するが，対話シェルや Linux コマンドが使用できないのはトラブルシュートの難易度が上がってしまう．そのため，Kubernetes の機能の 1 つである Ephemeral Containers[12] のように，デバッグ用のツールが入っているコンテナからデバッグ対象のコンテナを操作するといった方法を検証する．

参考文献

[1] Cloud Native Computing Foundation: Kubernetes, The Linux Foundation (online), available from ⟨https://kubernetes.io/ja/⟩ (accessed 2021-05-01).
[2] IDC Japan : 2021 年国内コンテナ／Kubernetes に関するユーザー導入調査結果を発表，IDC Corporate USA（オンライン），入手先 ⟨https://www.idc.com/getdoc.jsp?containerId=prJPJ47597721⟩（参照 2021-05-01）.
[3] NVD: CVE-2014-9357, National Vulnerability Database (online), available from ⟨https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9357⟩ (accessed 2021-05-01).
[4] NVD: CVE-2019-5736, National Vulnerability Database (online), available from ⟨https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736⟩ (accessed 2021-05-01).
[5] Open Container Initiative: opencontainers/runtimespec, Linux Foundation (online), available from ⟨https://github.com/opencontainers/runtime-spec⟩ (accessed 2021-05-04).
[6] The gVisor Authors: gVisor, Google, inc. (online), available from ⟨https://gvisor.dev/⟩ (accessed 2021-05-02).
[7] Kata Containers: Kata Containers, Open Infrastructure Foundation (online), available from ⟨https://katacontainers.io/⟩ (accessed 2021-05-02).
[8] 季栄原田，哲夫半田，正樹橋本，英彦田中：アプリケーションの実行状況に基づく強制アクセス制御方式，情報処理学会論文誌，Vol. 53, No. 9, pp. 2130–2147 (2012).
[9] NVD: CVE-2016-5195, National Vulnerability Database (online), available from ⟨https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195⟩ (accessed 2021-05-01).
[10] Cloud Native Computing Foundation: CoreDNS: DNS and Service Discovery, Linux Foundation (online), available from ⟨https://coredns.io/⟩ (accessed 2021-05-04).
[11] Cloud Native Computing Foundation: etcd — A distributed, reliable key-value store for the most critical data of a distributed system, Linux Foundation (online), available from ⟨https://etcd.io/⟩ (accessed 2021-05-04).
[12] Cloud Native Computing Foundation: Ephemeral Containers, Linux Foundation (online), available from ⟨https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/⟩ (accessed 2021-05-02).