Vol.2016-IOT-35 No.10
Vol.2016-SPT-20 No.10
2016/9/24

オンプレミスで実現する業務効率化のための OSS 基盤環境構築

森 健人[1,a)]　松浦 知史[1,b)]　金 勇[1,c)]　友石 正彦[1,d)]

概要：近年 Slack，Github，Dropbox といったクラウドサービスをチーム内での業務効率化を目的として導入する組織が増えている。機微な情報を業務で扱う組織であっても，これらのサービスクローンとして開発されている OSS をオンプレミスで導入することで，機密データを組織内部で管理しつつほぼ同等のサービス環境を導入することが可能である。しかし，これらの OSS サービスを内部構築して適切に管理するために必要な運用負担は大きく，これを支払う余裕のない組織も多い。そこでこの運用コストの問題を解決する一つのアプローチとして外部からのアクセスの一元化，認証サービスの一元化，サービス管理技術の一元化の 3 点を軸とした基盤環境構築に取り組んだ。複数のサービス環境を運用する場合でも導入から維持管理，監視までを少ない運用コストで実現する OSS サービス基盤環境の構築過程を報告する。

1. はじめに

近年チーム内業務効率化のため，Slack や Dropbox，Github といったクラウドサービスを導入する組織は多い。しかし機微な情報を扱う組織では外部に機密情報の管理を委ねることができない等の理由からこれらのサービスを業務に導入することは困難である。その一方で，これらのサービスの多くにはほぼ同等の機能提供を目指したサービスクローンが OSS として開発されている。例えば Rocket.Chat[*1] や ownCloud[*2]，GitLab[*3] といった OSS を上記のクラウドサービスの代わりにオンプレミスで導入することは可能である。しかしこの場合もセキュリティ対策や運用コストなどの課題は残る。

そこで今回はオンプレミス OSS で業務効率化ツールを導入するケースにおいて，複数サービス運用時もメンテナンスやログ監視に柔軟に対応でき，なおかつ構築・運用コストを抑えながらも機微な情報の適切な管理ができる環境構築に取り組んだ。ここでは具体的な手順と工夫を整理しながら，同様の環境構築を目指す際の一助となることを目的とした構築過程の報告を行う。

[1] 東京工業大学 Tokyo Institute of Technology, Meguro, Tokyo 152–8550, Japan
[a)] mori@cert.titech.ac.jp
[b)] matsuura@gsic.titech.ac.jp
[c)] yong@gsic.titech.ac.jp
[d)] tomoishi@noc.titech.ac.jp
[*1] https://rocket.chat/
[*2] https://owncloud.org/
[*3] https://gitlab.com/

2. OSS 基盤環境の要件と設計

チーム内コミュニケーションツールやコラボレーションツールの多くは，クラウドサービスとして展開され多くの企業で既に業務利用されている。端末を選ばないというクラウドの長所は業務効率化ツールへのニーズと合致しているものの，機微な情報を扱う組織では以下のような制約が導入への障害となる。

制約 A　機密データを外部に保持できない
制約 B　外部にサービスの管理を委託できない
制約 C　サービス運用に多くのコストを支払えない

例えば組織内 CSIRT のような機微な情報を扱う小規模組織で業務効率化ツールを導入することを考える。チームによっては扱うデータの機密性が高いにもかかわらずこれを外部に管理を委任できる権限を持っていない。そのため業務データを外部管理するようなツールの導入は難しい（制約 A）。これらのツールをオンプレミスで導入した場合，同様の理由でツールが扱うデータの機密性を適切に担保する必要がある。そのためツールとデータの管理を外部に委託できない（制約 B）。また，このようなツールを内部開発したり，管理責任ごと委託するような運用コストを支払うこともできない（制約 C）。以上の制約からオンプレミス OSS という形で導入する必要が生じる。この 2 章では上記の制約のもとで，オンプレミスで OSS ツールを導入する場合の要求要件を整理しそれに見合う構成を提案する。

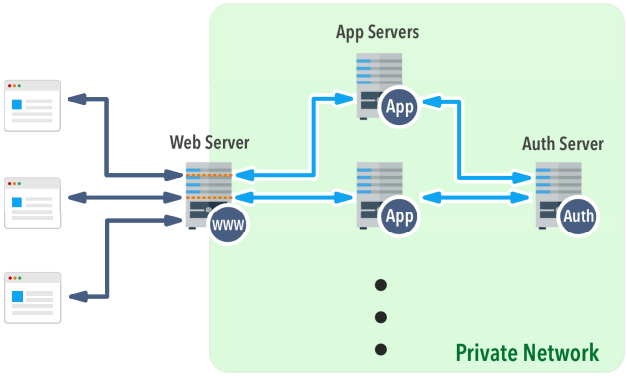

図 1 OSS 基盤環境の構成概要

2.1 要求要件

前項で挙げた制約のうち，制約 A・B についてはオンプレミスかつ OSS で Web サービス環境を構築することで満足する。しかしこの場合，機微な情報を扱うことからアクセスコントロールやログ監視などをはじめ適切なセキュリティ対策を打つ必要がある。さらに複数の Web サービスを運用する場合は，それぞれに対して運用コストが生じるため，制約 C の問題が導入への大きな障害となる。

上記のような環境でセキュリティ上求められる最低限の要件として以下の 3 つに着目する。

要件 a 外部からのアクセスを監視できる
要件 b 認証プロセスを監視できる
要件 c 常に最新環境を維持できる

以上の要件を押さえることで制約 A・B を満足させつつ，これを踏まえて少ない運用コストで実現することで制約 C に対処することを考える。さらに複数の OSS サービスを立ち上げることを前提にして運用コストを抑えることを念頭に置き，設計を行う。

2.2 OSS 基盤環境の構成と設計

前項で挙げた要件を満たす構成例を提案する。この 3 つの要求要件を，複数サービス運用も踏まえて少ない運用コストで実現することを目指す。要件 a に対して外部からのアクセスの一元化，要件 b に対して認証サービスの一元化，要件 c に対してサービス管理技術の一元化という 3 点を軸に設計方針を整理する。提案する構成の概要を図 1 に示す。

2.2.1 外部からのアクセスの一元化

2.1 項では要件 a として外部からのアクセスを監視できる必要性を挙げた。機密データを保持したサーバを守るためには厳密なアクセスコントロールが求められるが，複数のサービスを運用する場合でも少ない運用コストでこれを実現するために外部からのアクセスの一元化に取り組む。

複数の OSS サービスごとに Web サーバを管理・運用することはセキュリティ上好ましくない。制御すべき Web サーバが分散すると，人為的な設定ミスの誘発や異なるフォーマットのログを管理する等の手間を招く。また OSS サービスによっては Web サーバの種類を選べないケースも多く，適切な管理を行うためには設定方法などの学習コストも増大しがちである。これらの問題を解決するために運用する全てのサービスの窓口を統一制御する Web サーバを設置する。これは Web サーバアプリケーションのリバースプロキシ機能を用いて実現する。

インターネットへ露出するサーバはこのリバースプロキシを設定した Web サーバのみであり，各サービスへは必ずこの Web サーバ経由でのみアクセスさせる。また各サービスはこの Web サーバと管理セグメント以外から隔絶した環境に置く。その結果すべてのサービスへのグローバル環境からの通信は必ずこの Web サーバのアクセスログに記録される。

この構築による狙いは以下が挙げられる。

- Web サービスを運用する各サーバはプライベートネットワーク内に設置できる
- 各 Web サービスへの外部から通信を窓口となる Web サーバで一元的に制御できる
- 外部との通信ログはこの Web サーバのアクセスログとして集約できる

この構成では各 Web サービスが実行されているサーバには外部から直接アクセスすることができないため，設定ミス等による機微な情報が漏洩する危険性が少ない。また窓口と本体を分離したことで，サーバ障害時に Web サービスの仮想マシン自体を入れ替えるなどの対応を取りやすい。この窓口となる Web サーバの設定はリバースプロキシ配下にある全ての Web サービスへ反映されるため，IP 制限やアクセス認証，SSL 化などを一元的に設定することが可能である。加えて窓口となる Web サーバのアクセスログを監視するだけで，全ての Web サービスへの攻撃を検知できる。

2.2.2 認証サービスの一元化

2.1 項の要件 b では認証プロセスを監視できる必要性に着目した。機微な情報を守るためには意図しないログインや不正なログイン試行などを検知するための監視が必要となる。複数の OSS サービスを同時に運用した場合に少ない運用コストでこれを実現するために認証サービスの一元化に取り組む。

複数の Web サービスを運用する場合，通常はそれぞれのサービスごとにユーザアカウントを作成し管理することが必要になる。しかし運用する Web サービスが増えると，アカウント情報を分散管理する手間が増え，その結果ユーザと管理者の双方で制御しきれないアカウントが増加することとなり，これはセキュリティ上好ましくない。そこでアカウントの統一制御のため独立した認証サーバを設置する。

プライベートネットワーク内にこの認証サーバを設置

し，各 Web サービスへのログイン許可を認証サーバでのみ制御する。ユーザのアカウント情報は認証サーバで管理され，追加や変更を始め一時的なアカウントロックなどは認証サーバ側で制御する。また認証サーバは各 Web サーバと管理セグメント以外から隔絶した環境に置く。

この構築による狙いは以下が挙げられる。

- 全てのサービスのアカウント情報を一括管理できる
- 全てのサービスの認証ログを集約できる
- パスワードポリシー設定等を一括制御できる

各 Web サービスのログイン履歴などの情報は認証サーバに集約される。これにより不審な認証要求の検知を認証サーバのみ監視することで実現出来る。同時に不審なログイン試行を検知した場合の対応を一元的に設定することが可能となる。これは利用ユーザ側からの利点も多い。各ユーザは Web サービスごとにアカウントを作成する必要はなく，新規に導入されたサービスの利用がし易い。またユーザ情報の変更は認証サーバ内のものを更新するだけで全てのサービスに反映させることができる。一方で管理者側では，ユーザアカウントの追加や失効が容易にできるため管理コストの減らすことができ，ユーザグループごとにアクセス領域を分けるなどの情報の格付けにも対応し易い。

2.2.3 OSS サービスの管理技術の一元化

2.1 項では要件 c としてサービスを最新環境へ維持する必要性を挙げた。機微な情報を守るためにはサービスとそれを運用する基盤環境を最新状態へ維持し続けることが求められる。複数の OSS サービスを同時に立ち上げた場合に少ない運用コストでこれを実現するために OSS サービス管理技術の一元化に取り組む。

成熟した OSS サービスは少なく，ベースとなる言語や DB を導入側が選べないケースが多い。管理する Web サービスが増えるほど維持管理コストは増大するが，それぞれの基幹システムの構築・維持に必要な学習コストなどを含めるとこの運用コストは膨大となる。この問題に対処するため各 Web サービスは仮想コンテナとして導入する。各サービスをコンテナ単位で管理することで基本的な管理技術の一元化を行う。

全ての Web サービスは直接サーバの OS へインストールするのではなく，コンテナとして各サーバへ導入する。必要ならば DB や Web サーバもそれぞれコンテナとして同一サーバに展開し，各 Web サービスに必要なコンテナ群をまとめて管理する。新規導入やバックアップ，アップデートなどのサービス管理はコンテナ単位で行う。

この構築による狙いは以下のように整理できる。

- Web サービスの依存システムを一括導入・管理できる
- コンテナごとに容易に更新・復旧ができる
- コンテナ制御のコマンドのみでメンテナンスできる

コンテナとして Web サービスを導入することで，それぞれが依存する言語や DB，OS といった基盤環境を気にする必要がなくなる。提供されている OSS サービスの中には既にコンテナイメージとして管理されているものも多く，その場合は目的に応じたコンテナを，自分たちが管理しやすい OS へ展開するだけで良いためメンテナンス性に優れる。開発者コミュニティによりメンテナンスされているコンテナであれば，コンテナを丸ごと入れ替えるだけで容易に最新環境へと移行することができる。これには用意されたコマンドのみで実行可能であり，基本的なコンテナ維持に必要な学習コストは低い。また一つのサービスに対して必要なコンテナセットをまとめて管理することができるため，依存関係の問題が発生するリスクが低い。

加えて，コンテナのサーバ間の移行が容易であることや同一サーバに複数コンテナ・サービスを展開することが可能であることから，サービス管理における柔軟性も高い。

3. OSS 基盤環境の実装

前章の設計を踏まえて，ここでは今回取り組んだ実装について具体的に整理する。手順については必要に応じて付録にて詳細を述べる。

3.1 基本構成

ここでは構成の概要を記し，後述する各項にて詳細を述べる。各サーバは VMware vSphere ESXi 5.5 をホストとして，その上に仮想マシンとして Ubuntu 14.04 Server を利用することを基本とする。OSS 基盤環境は我々が用意した仮想化基盤上 [1] に展開する。

窓口となる Web サーバ，認証サーバ，各 Web サービスはそれぞれ異なる仮想マシンで構築する。また管理のし易さを考えて各 Web サービスは依存するコンテナセットをそれぞれ一つの仮想マシン上に展開する。下記にそれぞれベースとする仮想マシンのスペックを示す。

- 使用する仮想マシンのスペック
  - VMWare 仮想マシン Ver. 10
  - CPU：1 Core
  - MEM： 8GB
  - HDD：1TB（Thin Provisioning）

2.2 項で示した設計を実現するため，Nginx によるリバースプロキシ環境，OpenLDAP による認証基盤環境，Docker コンテナによる Web サービス環境に取り組む。今回の構築で採用したそれぞれのソフトウェアの詳細を表 1 に記す。

表 1 導入したソフトウェア

ソフトウェア	バージョン	URL
Nginx	1.10.1	https://nginx.org/
OpenLDAP	2.4.31-1	http://www.openldap.org/
Docker	1.10.3	https://www.docker.com/

また今回は業務効率化ツールとして代表的な OSS である Rocket.Chat，GitLab，ownCloud を Docker コンテナとし

表 2　導入した OSS

OSS	バージョン	Docker イメージ（Docker Hub）
Rocket.Chat	0.37	rocketchat/rocket.chat:0.37
GitLab	0.8.10	sameersbn/gitlab:0.8.10
ownCloud	9.03	sameersbn/owncloud:9.0.3

て導入する。具体的なバージョンと Docker イメージの参照元を表 2 に示す。ここで示す Docker イメージは Docker Hub[*4] にて公開されている。

ここでは割愛するが，Docker 環境構築をはじめとした後述のサーバ構築は全て Ansible のプレイブックにより管理し，必要に応じて一括構築を行った。これにより基盤環境構築の管理コストの軽減を実現している。

3.2　Nginx によるリバースプロキシ環境構築

2.3.1 で述べたグローバルアクセスの一元化のため，リバースプロキシの機能を持つ Nginx を窓口となる Web サーバとして導入する。

高機能，高パフォーマンスでリバースプロキシとして導入実績があるという理由で Nginx を採用した。また SSL リバースプロキシとしての利用可能な点や，リバースプロキシでも WebSocket が安定利用が確認できていた点も採用した大きな理由である。

Nginx は執筆時の最新安定版であるバージョン 1.10.1 を使用する。Ubuntu 14.04 の初期リポジトリからでは導入することができないため，公式リポジトリを追加する必要があることに注意する。

今回構築するリバースプロキシ環境を図 2 に示す。

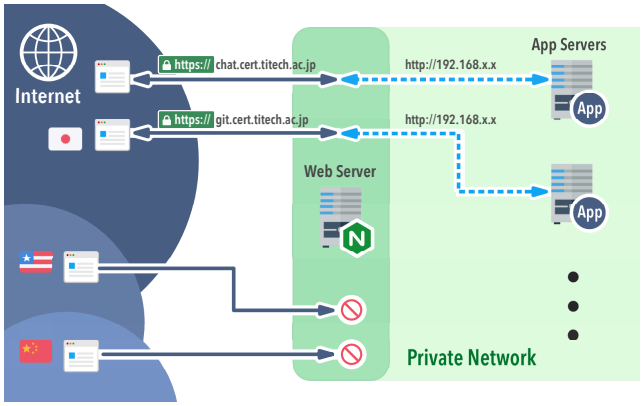

図 2　リバースプロキシ環境の構成

3.2.1　Nginx の導入

各 Web サービスで使用するドメインを Nginx サーバの CNAME として DNS に登録する。そしてリバースプロキシによって各 Web サービスが実行されている仮想マシンの IP アドレスに振り分ける。

具体的なリバースプロキシ環境の構築については特殊なことは行っていない。しかしバックエンドで利用する Web サービスによっては WebSocket 中継の有効化，リクエストボディの最大サイズの変更やプロキシタイムアウトの設定などを行う必要があった。

またこの Nginx のサーバに SSL 証明書を用意し，これを用いて各 Web サービスの外部通信の HTTPS 化を実現している。これにより各 Web サービスごとに SSL 証明書を取得・管理しなくとも，窓口の Web サーバで一元管理を可能にしている。Nginx から各 Web サービス本体の仮想マシンまでは非暗号化通信が使用されるが，その点は vSphere の仮想ネットワーク内のみ，もしくは管理セグメント内での通信に限るため問題ないと判断できる。

またこの環境では Nginx の設定ファイルを適宜編集するだけで，各 Web サービスのアクセスコントロールを一括で設定・管理できる。実際の運用では，GeoIP を用いた国別アクセス制限や特定 IP のみのアクセス制限などを行っている。これについては付録 A.1 にて詳細を述べる。

3.2.2　アクセスログの監視と検知

リバースプロキシを用いて外部への窓口を一元化したことにより，各 Web サービスへの外部からのアクセスは全て Nginx のアクセスログとして記録される。つまりこのアクセスログを監視するだけで全ての Web サービスに対する不審なアクセスの検知を行うことができる。

よって fail2ban[*5] などを利用して不審なアクセスの検知と遮断を自動化したい場合、検知と遮断のためのシステムをこの Nginx の仮想マシンで完結させることができる。詳しい設定方法は割愛するが、これらのツールを用いることで各 Web サービスに対する過剰な不正リクエストなどを窓口である Nginx サーバのみで遮断することが可能である。一方で Splunk を始めとする統合ログ解析ツール等このアクセスログを監視・管理する環境が整っている場合は，後述する OpenLDAP のログ等と連携させてアクセス解析などが行うことでよりより精度の高い検知を行うこと s できる。

上で示したように，Nginx によるリバースプロキシ環境の導入によってグローバルアクセスの一元化を実現できる。これにより外部とのアクセス制限を一括設定や，ログ監視・検知が容易に行える環境が整う。そしてこれは複数サービスを導入するようなケースで管理コストの大きな軽減を見込むことができる。

3.3　OpenLDAP による認証環境構築

2.3.2 で述べた認証サービス一元化のために OpenLDAP サーバを導入する。

LDAP の他に様々な認証サーバシステムは存在するが，導入実績が多くシンプルに構築が可能な LDAP を利用す

*4　https://hub.docker.com/

*5　http://www.fail2ban.org/

る。今回導入するようなメジャーな OSS サービスの多くが LDAP 認証に対応しているおり，認証の一元化という点で技術的に枯れている LDAP を採用することのメリットは大きい。そこで OSS として提供されて導入実績もある OpenLDAP を採用する。

3.3.1 OpenLDAP の導入

運用する全ての Web サービスと連携する一つの認証サーバを OpenLDAP により構築する。これは認証サーバとして用意した仮想マシン上に直接インストールされている。このマシンは管理セグメントの他は各 Web サービスの仮想マシンからのアクセスのみが可能にしたプライベートネットワーク内に配置している。

また，機微な情報を扱うことから構築に際に留意するべき点を以下に挙げる。

匿名バインドの禁止

初期状態のままではバインドなしで外部から ldap プロトコルを用いてユーザ情報にアクセス可能となっている。このままだと誤って OpenLDAP のポートが開放された場合に LDAP のエントリ情報が誰でも閲覧可能となる。これを防ぐために匿名バインドによるユーザ検索を禁止する設定が必要となる。また同時に各 Web サービスでのユーザ認証には検索権限のみ付与された専用のユーザを利用させる。この手順については付録 A.2 で述べる。

パスワードポリシーの設定

OpenLDAP では ppolicy オーバーレイを用いることでパスワード制御に関する各種設定を行うことができる。これを用いてパスワードの組み合わせ・文字数などの基本設定を一括設定する。また一定回数以上の認証失敗時に該当のアカウントを一定時間ロックさせるように設定することで，不正なログイン試行でパスワードが暴かれるのを防ぐ。この ppolicy オーバーレイを有効にする手順については付録 A.3 で述べる。

3.3.2 認証ログの監視と検知

OpenLDAP を用いて認証システムを一元化したことにより，全ての Web サービスに対するログインリクエストの情報が OpenLDAP のログへ集約される。これによりインシデント発生時など早急にユーザのログイン情報を解析したい場合にはこのログを調べるだけで良いことになる。ただしデフォルト環境ではログの出力がされないため有効化する必要がある点に注意する。

OpenLDAP では特定のユーザへの不正なログイン試行を検知して該当アカウントを一時的にロックすることができるが，これには誤ったユーザ名に対するログイン試行を防ぐ機能はない。また Web サービスのログインリクエストが WebSocket 通信を用いる場合，ログイン試行自体が Nginx のアクセスログに残らない。そのため過剰なログイン試行を防ぐには OpenLDAP の認証ログを監視する必要がある。対処としては認証ログを監視して一定時間内に一定回数以上のログイン試行があった場合にアラートを出す。今回はこのアラートを Rocket.Chat を経由して全ユーザに通知させるように連携させている。このような手間はアクセスログ，認証ログの一元化によって収集できる情報量が減ったことに起因する。これについては今後の課題として後述する。

上で示したように OpenLDAP による認証サービスの一元化により，内部からの機密データへのアクセス状況を把握することが容易となる。アカウント情報の制御や認証制御の一括設定が可能となり，LDAP 認証に対応してさえいればユーザアカウントを気にすることなく新規 Web サービスの導入などを行うことができる。これにより Web サービスの導入・廃止，ユーザアカウントの追加・削除の際に生じる設定コストが大幅に削減されることが見込まれる。

一方でユーザ情報は OpenLDAP サーバで統一管理されるため，基本的には各 Web サービスで用意されているパスワード変更機能を利用することができない。このため初期パスワードの変更など，各ユーザが自分の情報のみを編集できる窓口が必要となる。今回は LDAP サーバ上の個人情報を操作できる簡易な Web アプリケーションを NodeJS を用いて作成し，これに対処した。これにより各ユーザがログインしてパスワードなど最低限の情報を自分で変更ができる環境を提供している。

3.4 Docker コンテナによる Web サービス構築

2.2.3 項で述べた OSS 管理手法の一元化のため，Docker コンテナによる Web サービス構築を行う。

コンテナ型仮想化技術を利用するメリットについては既に述べた通りである。そこで既に導入実績のある Docker を採用する。最近では OSS サービスの多くも Docker コンテナによる導入をサポートしており，その開発者がコンテナイメージを整備しているサービスも多い。Docker コンテナによる Web サービス構築のイメージを図 3 に示す。

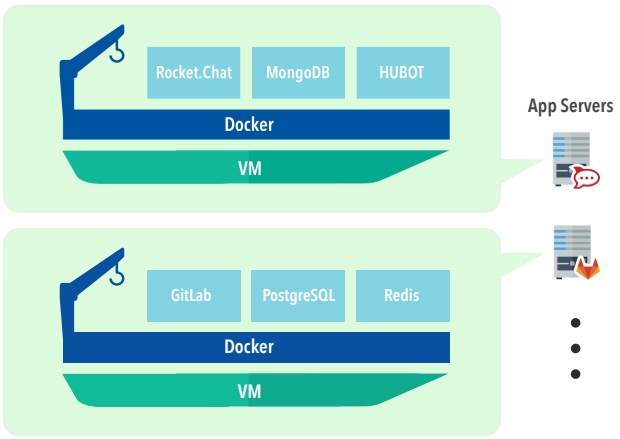

図 3 Docker コンテナによる Web サービス構築

3.4.1 Docker 環境の導入

運用する各サービスは全て Docker コンテナとして導入し，管理を容易にするためサービスに必要なコンテナセットをそれぞれ一つの仮想マシン上に展開する。各サービスは窓口となる Nginx サーバからのみアクセス可能となっている。

まずは核となる Web サービスを Docker コンテナとして作成する。コンテナの作成には Docker のラッパーツールである Docker Compose[3] を用いて手順を単純化させる。これを用いるとコンテナ構成を記述した YAML ファイルをベースにコンテナセットを容易に構築・管理することができる。コンテナの作成には Docker Hub 等にて共有されているコンテナイメージを指定して容易に構築が可能である。開発チームが管理する dockerfile が提供されていない場合は有志でメンテナンスされているものを利用するが，この場合は OSS のメンテナンスは開発チームと dockerfile の提供元に依存することになる点に注意が必要である。具体的なコンテナ構築手順については割愛するが，一例として Rocket.Chat 環境を構築する docker-compose.yml の作成例について付録 A.4 に示す。

また機微な情報を扱うことからコンテナに対するアクセス制御を適切に設定する必要があるが，Ubuntu14.04 で標準として用意されているファイアーウォール設定ツールである ufw と Docker を併用する際には留意すべき設定項目がある。これについては付録 A.5 にて詳細を述べる。

上で示した構築では Docker を使用する各仮想マシン上で，Docker と Docker Compose の導入と上記の ufw の設定を行う手間を避けられない。この構築コストを削減するために一連の操作のための Ansible のプレイブックを作成し，構築の自動化を行った。このプレイブックの設定例の一部を付録 A.6 にて述べる。

3.4.2 OSS のアップデート

OSS の多くは未成熟でありそれゆえにアップデートの頻度が高い。必ずしも最新リリースを利用する必要はないが，サービスによっては毎週アップデートがありそれぞれが大幅な機能改善がもたらすようなサービスも多い。この理由から OSS においてはセキュリティ対策だけでなく機能改善という目的でバージョンアップを行う価値があると言える。

Docker コンテナによる導入によって，このアップデートの手間は大幅に削減される。仮想コンテナは基本的に独立したモジュールとして導入するため，アップデートの際は古いコンテナから新しいコンテナへすげ替える作業を行うだけで良い。これは docker-compose.yml による環境構築を行った場合は，docker-compose pull && docker-compose up -d など Docker Compose の基本コマンドのみで実現できる。ただし docker-compose.yml にて正しくコンテナイメージのバージョンが指定されていることと，そのコンテナイメージが適切にメンテナンスされている必要がある点に注意する。またコンフィグや DB のデータ領域などは予めコンテナ外へ保持するよう適切に設定する必要がある。

上で示したように Docker コンテナによる Web サービスの導入によって OSS サービスの維持・管理に必要な運用コストの軽減が実現できる。これは複数の OSS サービスを運用し易くするのに加え、サービスのアップデートのし易い点でセキュリティ向上に貢献している。

今回は各 Web サービスをそれぞれ異なる仮想マシン上の Docker コンテナとして展開したが必ずしもこのようにする必要はない。環境・設備に応じてサーバマシン上に直接複数コンテナを展開することも検討する必要がある。

4. おわりに

機微な情報を扱う組織で運用コストを抑えながらも安全性を考慮した OSS 基盤環境構築を行った過程を報告した。さらに今回の構築例ではアクセスと認証の窓口を一元化した結果，Web サービスの新規導入やメンバー構成の変更などに柔軟に対応出来る OSS 基盤環境を実現できた。

一方で今回示した例は少ない運用コストで最低限の不正アクセス対策を備える構成ではあるが，インシデント発生時に状況を解析するには結局各々の Web サービス内のログを追う必要があり，事後対応環境が良いとは言い難い。リスクと対応環境構築のコストを鑑みる必要があるが，必要な場合には外部との窓口 (Nginx) と認証の窓口 (OpenLDAP) のみでも詳細なログを集積できるような柔軟性なシステム構成を探る必要があると考える。

謝辞 本研究は JSPS 科研費 15K00115 の助成を受けたものである。

参考文献

[1] 松浦 知史, 森 健人, 金 勇, 友石 正彦：拡張性を考慮した小規模仮想化基盤の構築, 情報処理学会研究報告 (2016.03).
[2] OpenLDAP リファレンス (online), 入手先 ⟨http://www.openldap.org/doc/admin24/⟩ (2016.8.27).
[3] Docker Compose リファレンス (online), 入手先 ⟨https://docs.docker.com/compose/⟩ (2016.08.27).

付　　録

A.1 GeoIP による国別アクセス制限（Nginx）

Nginx の http_geoip_module を用いると IP アドレスからアクセス元の国名を判別し，国ごとにアクセス制限などを容易に設定することができる。(nginx -V でモジュールがロード済みか否かを確認できる。）日本国内アクセスのみを許可する設定例を以下に示す。

まずはアクセス元 IP が日本であった場合, allowd_countey 変数に yes を定義するように/etc/nginx/nginx.conf の http

ディレクティブに以下のような設定を加える。

```
    http{
...
     geoip_country /usr/share/GeoIP/GeoIP.dat;
     map $geoip_country_code $allowed_country {
       default no;
       JP yes;
     }
     ...
```

あとはそれぞれの仮想サーバに対して該当の変数が yes でない場合に国外からのアクセスに対する挙動を設定すれば良い。

この場合プライベート IP は国外扱いとなるため，許可したい場合は別途設定が必要である点に注意する。

A.2 匿名バインドの禁止（OpenLDAP）

OpenLDAP を導入して初期状態のままでは匿名での ldapsearch が実行可能になっている。提案した環境では OpenLDAP サーバへアクセス可能な端末を絞っているものの，安全のためこれを禁止する。そして検索には専用のユーザを作成し，各 Web サービスからの ldapsearch はこのユーザにバインドして行う。この設定手順を以下に示す。

まず設定を記した config-disallow_anon.ldif を作成し，これを OpenLDAP に読み込ませる。

```
$ cat << EOF > config-disallow_anon.ldif
dn: cn=config
changetype: modify
add: olcDisallows
olcDisallows: bind_anon

dn: cn=config
changetype: modify
add: olcRequires
olcRequires: authc

dn: olcDatabase={-1}frontend,cn=config
changetype: modify
add: olcRequires
olcRequires: authc
[EOF]

$ sudo ldapadd -Y EXTERNAL -H ldapi:/// -f config-disallow_anon.ldif
```

適切に設定されると，匿名での ldapsearch は以下のようなエラーが表示される。

```
$ ldapsearch -H ldap://ldap.cert.titech.ac.jp:389 -x  -LLL -b
dc=cert,dc=titech,dc=ac,dc=jp -s sub '(uid=ldapuser)'
Server is unwilling to perform (53)
Additional information: authentication required
```

匿名バインドを禁止した状態では cn=admin（管理者）でバインドしてパスワードを入力することで検索することが可能である。ただし各 Web サービスで管理者のパスワードを保持しておくことは危険であるため，検索のみ許可された cn=searcher ユーザを作成する。

```
$ cat << EOF > searcher.ldif
dn: cn=searcher,dc=cert,dc=titech,dc=ac,dc=jp
objectClass: simpleSecurityObject
objectClass: organizationalRole
cn: searcher
[EOF]

$ echo userPassword: $(slappasswd -s searcher-no-password) >> searcher.ldif
$ ldapadd -x -D cn=admin,dc=cert,dc=titech,dc=ac,dc=jp -W -f searcher.ldif
```

最後に各 Web サービスの LDAP 連携設定でこの検索専用ユーザをバインドユーザとして設定しておく。

A.3 パスワードポリシーの設定（OpenLDAP）

OpenLDAP では ppolicy オーバレイを用いることでパスワード設定・認証に関する要件を課すことができる。これは ppolicy モジュールとして用意されており，モジュールのロード，オーバレイの設定，スキーマの設定，設定用 DN 作成という手順を踏む必要がある。以下に作業の手順を整理する。

1. モジュールをロードする

まずは ppolicy.la をロードする。これは OpenLDAP のデフォルトのパッケージに含まれている。

```
$ cat << EOF > config-ppolicy.ldif
dn: cn=module{0},cn=config
changetype: modify
add: olcModuleLoad
olcModuleLoad:ppolicy.la

$ sudo ldapmodify -Y EXTERNAL -H ldapi:/// -f config-ppolicy.ldif
```

2. スキーマを追加する

openldap ではスキーマと呼ばれるデータ定義ファイルを用いることで，そのオブジェクトクラスを構成する属性群が定義される。今回は最初から用意されている ppolicy 用スキーマ (/etc/ldap/schema/ppolicy.ldif) をそのまま利用するが，必要に応じて適宜編集することが可能である。

```
$   sudo ldapmodify -Y EXTERNAL -H ldapi:/// -f /etc/ldap/schema/ppolicy.ldif -a
```

3. オーバーレイを設定する

ppolicy オーバーレイの基本設定を ldif ファイルに記述し，ロードする。今回は olcPPolicyUseLockout のみ真（ロックされたら invalidCredential を返す）に設定し，デフォルト設定用 DN として，cn=passwordDefault を指定する。

```
$ cat << EOF > ppolicy.ldif
dn: olcOverlay=ppolicy,olcDatabase={1}hdb,cn=config
objectClass: olcOverlayConfig
objectClass: olcPPolicyConfig
olcOverlay: ppolicy
olcPPolicyDefault: cn=passwordDefault,ou=Policies,dc=cert,dc=titech,dc=ac,dc=jp
olcPPolicyHashCleartext: FALSE
olcPPolicyUseLockout: TRUE
olcPPolicyForwardUpdates: FALSE

$ sudo ldapmodify -Y EXTERNAL -H ldapi:/// -f config-ppolicy.ldif
```

4. ポリシーを設定する

一つ前の手順で指定した cn=passwordDefault のエントリーに記述した pwd*がパスワードポリシーとして反映される。

```
$ cat << EOF > passwordDefault.ldif
#
dn: ou=Policies,dc=cert,dc=titech,dc=ac,dc=jp
ou: Policies
objectClass: organizationalUnit

dn: cn=passwordDefault,ou=Policies,dc=cert,dc=titech,dc=ac,dc=jp
objectClass: pwdPolicy
objectClass: person
objectClass: top
cn: passwordDefault
sn: passwordDefault
pwdAttribute: userPassword
pwdCheckQuality: 0
pwdMinAge: 0
pwdMaxAge: 0
pwdMinLength: 8
pwdInHistory: 5
pwdMaxFailure: 3
pwdFailureCountInterval: 0
pwdLockout: TRUE
pwdLockoutDuration: 0
pwdAllowUserChange: TRUE
pwdExpireWarning: 0
pwdGraceAuthNLimit: 0
pwdMustChange: FALSE
pwdSafeModify: FALSE
EOF

$ ldapadd -x -D cn=admin,dc=cert,dc=titech,dc=ac,dc=jp -W -f passwordDefault.ldif
```

このように OpenLDAP では ldif ファイルをロードすることでリアルタイムに設定を反映することができる。

ここでは ppolicy モジュールの導入例について簡単に示

したが，他に access/audit モジュールや monitor モジュールなど，LDAP の挙動の詳細に設定する機能が用意されており，環境に応じて設定を行う必要がある。

A.4 Docker Compose によるコンテナ構築例（Docker）

ここでは Rocket.Chat のコンテナ環境を構築する docker-compose.yml の設定例を記す。

Docker Compose とは Docker のラッパーツールであり，コンテナ構成を記述した YAML ファイルを用意するだけで Docker のコンテナセットを容易に構築・管理することができる。

Rocket.Chat については開発チーム自身がコンテナイメージをメンテナンスしているため，これを用いることでバージョンアップ等も含めて容易に管理が可能になっている。

以下が Rocket.Chat に必要なコンテナセットを構築する docker-compose.yml の一例である。

```
$ car << EOF > docker-compose.yml
mongo:
  restart: always
  image: mongo
  volumes:
     - ./data/runtime/db:/data/db
     - ./data/dump:/dump
  command: mongod --smallfiles --oplogSize 128

rocketchat:
  restart: always
  image: rocketchat/rocket.chat:latest
  volumes:
     - ./uploads:/app/uploads
  environment:
    - PORT=3000
    - ROOT_URL=http://192.168.x.x:3000
    - MONGO_URL=mongodb://mongo:27017/rocketchat
  links:
    - mongo:mongo
  ports:
    - 3000:3000
[EOF]
```

上記の docker-compose.yml を作成した上で，docker-compose up -d を実行する。これにより Rocket.Chat とそれに必要となる MongoDB の Docker コンテナが作成・実行される。

同様の手順で GitLab や ownCloud なども Docker Hub にて共有されているコンテナイメージを使用して容易に環境構築することが可能である。

A.5 ufw を有効にする（Docker）

Ubuntu 14.04 では ufw コマンドを用いて容易にファイアーウォールを設定することができる。今回利用した Docker のバージョン 1.10.3 では，Docker デーモン起動時に iptables を書き換えて仮想コンテナをブリッジネットワークに接続する。その結果ホストの ufw の設定を Docker コンテナに反映させるには，設定を加える必要がある。この手順について以下に整理する。

1. Docker による iptables の編集を防ぐ

まずは iptables を自動設定しないオプションを有効にする。これには/etc/default/docker を以下のように編集する。

```
$ sudo vi /etc/default/docker
...
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
DOCKER_OPTS="--iptables=false"
...
```

2. ufw のパケット転送を許可する

次に ufw でパケット転送を有効にする設定を行う。これには/etc/default/ufw を以下のように編集する。

```
$ sudo vi /etc/default/ufw
...
DEFAULT_FORWARD_POLICY="ACCEPT"
...
```

3. 仮想ブリッジのルーティングを行う

最後に Docker コンテナが接続される仮想ブリッジのルーティング設定を加える。これは ufw 起動時に実行されるように/etc/ufw/before.rules を以下のように編集する。

```
$ sudo vi /etc/ufw/before.rules
...
*nat
:DOCKER - [0:0]
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
COMMIT
*filter
:ufw-before-input - [0:0]
:ufw-before-output - [0:0]
:ufw-before-forward - [0:0]
:ufw-not-local - [0:0]
# End required lines
...
```

上記の設定が完了したら忘れずに仮想マシンを再起動して iptables の再構築後にも適切にファイアーウォールが設定されているか確認する。

A.6 Ansible による Docker 環境構築例（Docker）

ここでは Ansible を用いた Docker 環境構築のためのプレイブックの一例を記す。

本論で示した環境構築では適宜仮想マシン上で Docker 環境を構築する手間が必要となる。そこで複数サービスの構築コスト削減のために Ansible を用いたインフラ環境構築の自動化を行う。Docker の基本環境構築を行うプレイブックについては Ansible Galaxy[*6] にて数多く公開されているため，ここでは A.5 で示した設定を自動化するためタスク例のみ抜粋して以下に示す。

```
### Docker のオプション設定
- name: set --iptables=false to /etc/default/docker
  lineinfile: >-
     regexp="^DOCKER_OPTS=
     line=DOCKER_OPTS="--iptables=false"
     dest=/etc/default/docker
     state=present
     insertafter="^#DOCKER_OPTS=

### ufw 用の NAT 設定を行う
- name: insert NAT conf to /etc/ufw/before.rules
  blockinfile:
     dest: /etc/ufw/before.rules
     insertbefore: \^*filter
     block: |
        *nat
        :DOCKER - [0:0]
        -A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
        -A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
        -A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
        COMMIT

- name: set DEFAULT_FORWARD_POLICY="ACCEPT" to /etc/default/ufw
  lineinfile: >-
     regexp="^DEFAULT_FORWARD_POLICY=
     line=DEFAULT_FORWARD_POLICY="ACCEPT"
     dest=/etc/default/ufw
     state=present
     insertafter="^#DEFAULT_FORWARD_POLICY=
```

*6 https://galaxy.ansible.com/