情報ネットワークⅡ（情213）
【第10回】
ホストのセキュリティ
（教科書：第6章）

担当教員：長田智和

E-Mail: nagayan@ie.u-ryukyu.ac.jp

URL: http://n-lab.info/

（講義日：2018年12月6日）

第6章：ホストのセキュリティ

6.1 バッファオーバーフローの概要

- 6.1.1 バッファオーバーフローの仕組み
  - メモリ空間上でバッファーの境界を越えてデータが上書きされてしまうこと。
    - スタック領域で起きる場合はスタックオーバーフロー
    - ヒープ領域で起きる場合はヒープオーバーフロー
  - バッファーオーバーフローは、攻撃者が悪意のあるプログラムを実行するために用いられる。
    - 関数のリターンアドレスを巧妙に書き換える。
- （教科書p.153の図6.1を参照）

6.2 バッファオーバーフローを用いた攻撃例

- 6.2.1 ローカルでの攻撃
  - 目的：管理者権限レベルの取得
  - ネットワークサーバープログラムなど、管理者権限で動作しているプログラムが標的となる。
  - 直接的にターゲットとなるホストを操作する。
- 6.2.2 リモートからの攻撃
  - 目的：管理者権限レベルの取得
  - ターゲットホストから別ホストや攻撃元ホストに接続させ、shellコードの実行やマルウェアを実行する。
  - 間接的にターゲットとなるホストを操作する。
- （教科書p.154-155の図6.2-6.3を参照）

6.3 バッファオーバーフローの詳細と対策

- 6.3.1 プロセスのメモリ配置とスタック
  - プロセス実行時のアドレス空間
    - 実行コードがテキストセグメントに格納される。
    - 初期値を持つ静的変数・大域変数は、データセグメントに格納される。
    - 初期値を持たない静的変数・大域変数は、初期値0としてBSS（Block Started by Symbol）に格納される。
    - C言語のmalloc関数等で確保された領域は、ヒープ領域に格納される。
    - 最上位アドレスから若いアドレスに向けて、スタック領域が確保される（積まれる）。
- （教科書p.157-158の図6.4-6.5を参照）

6.3 バッファオーバーフローの詳細と対策

- 6.3.2 プログラム実行時のスタックの様子
  - （教科書の例で説明）
- （教科書p.158-159の図6.6-6.9を参照）

6.3 バッファオーバーフローの詳細と対策

- 6.3.3 スタックオーバーフローの仕組み
  - strcpy関数で変数のサイズを超えてデータをコピーし、ebpレジスタやリターンアドレス値を上書きする。
  - リターンアドレス先に不正なコードを用意してそれを実行させたり、正規のシステムコールを呼び出して実行（Return-to-libc攻撃）させることができる。
- （教科書p.160-162の図6.10-6.15を参照）

6.3 バッファオーバーフローの詳細と対策

- 6.3.4 ヒープオーバーフローの仕組み
  - スタックオーバーフローに比べれば悪用しにくい。
    - どのバッファーが攻撃対象になり得るか不確定であるため。
    - 攻撃者が書き込むバッファーが、攻撃対象のバッファーの下位になければならないため。
  - 攻撃対象のソースコードが閲覧可能で、重要なバッファー領域の下位アドレスに攻撃者が任意のデータを埋めることが前提となる。

6.3 バッファオーバーフローの詳細と対策

- 6.3.5 ヒープオーバーフロー攻撃の例
  - （教科書の例で説明）
- （教科書p.163-165の図6.16-6.19を参照）

6.3 バッファオーバーフローの詳細と対策

- 6.3.6 バッファオーバーフローを招く関数
  - 使ってはいけない関数（C言語の場合）:
    - gets() → fgets() を使う。
    - strcat() → strncat() を使う。等々。
  - C/C++言語は効率を重視しているため、バッファー境界チェックがない。→ その分、高速に動作する。
  - プログラマがバッファー境界に配慮する必要がある。
- （教科書p.165-167の表6.1,図6.20-6.23を参照）

6.3 バッファオーバーフローの詳細と対策

- 6.3.7 その他のバッファオーバーフロー対策
  - セキュリティパッチを適用する。
  - 低レベル言語（C/C++,アセンブラ等）ではなく、高級言語（Java, Perl, Ruby等）を使用する。
  - C言語では、セキュリティチェックをしてくれるコンパイラを用いる。
  - 比較的新しいOSでは、DEP（Data Execution Prevention）やASLR（Address Space Layout Randomization）を利用する。

何れも、万能ではありません。。。

6.4 セキュアOSとセキュアブート

- 6.4.1 セキュアOS
  - 強制アクセス制御・最小特権を実現したOS
    - 強制アクセス制御：操作する主体（ユーザー、アプリケーション）と操作される対象（ファイルやディレクトリなどのリソース）をレベル分けし、レベルに応じてシステムが強制的にアクセス権限を決定する方式 ⇔ 任意アクセス制御
    - 攻撃されたとしても、制御を奪われたプログラム以外への攻撃の被害が拡大する可能性が低くなる。
  - セキュアOSは普及がネック
    - 運用上の負担が大きい（設定が複雑で難しい）
    - AndroidなどLinux系のセキュアOSの導入は進んでいる。

6.4 セキュアOSとセキュアブート

- 6.4.2 Linuxにおけるセキュア OS
  - SELinux
    - ロールベースアクセス制御（RBAC）型
    - 権限を付与されていない操作は不可
  - AppArmor
    - パスベースアクセス制御型
    - 権限を付与されていない操作は制限なく実行可
  - TOMOYO Linux
    - パスベースアクセス制御型
    - ポリシーの自動学習機能によって導入が比較的容易
- （教科書p.170の図6.24を参照）

6.4 セキュアOSとセキュアブート

- 6.4.2 Linuxにおけるセキュアos
  - LSM（Linux Secure Modules）
    - Linuxカーネルに追加されたフレームワーク
    - 任意アクセス制御で権限チェックされたシステムコールを、さらにセキュアOSに権限チェックを問い合わせる。
- 6.4.3 セキュアブート
  - 許可されていないOSやドライバーが起動時に実行されないようにする仕組み。UEFI仕様の一部。
  - ホストの電源投入からマルウェア対策ソフトの起動までの検証を行う。
- （教科書p.171-172の図6.25-6.26を参照）

【次回予告】
第11回
ネットワークセキュリティ
（第7章）