IoT マルウェアの機能判別を目的とした静的解析と動的解析に基づく解析手法

与那嶺　俊[†]　門林雄基[†]

1. はじめに

マルウェア解析の手法は，一般的に静的解析と動的解析の 2 つに区分される．静的解析はマルウェアを直接実行せずマルウェアの実行ファイル中のコードを分析し処理を解明する作業である．一方の動的解析はマルウェアをシステム上で実行し，マルウェアが実行した API やシステムコール等を分析することで処理を解明する．動的解析はプログラムを実際にシステム上で実行するため多くの情報が得られる．本発表では，特定の処理を行うコードの解析における静的解析と動的解析を組み合わせた方法を述べる．それぞれの解析手法の利点を認識した上で併用することで，解析を自動化する手法の構築を目指す．単一の IoT マルウェア検体 (md5:fbb94cf7d98341e13dc1f6468f6e4e6b) を対象に解析を行い，C&C 機能の解析を行う．

2. 実験内容

2.1 システムコール監視

ネットワークから隔離された環境において検体を 2 分間実行し，検体が発生させたシステムコールを観測した．システムコールのモニタリングにはオープンソースツールの Triton[1] を用いて作成したプログラムで行った．Triton は DBI(動的バイナリ計装) をサポートする．DBI は命令の観測のみならずプログラム実行時におけるメモリやレジスタ内のデータの書き換えにも使用できる．処理以下に観測したシステムコールの一覧を左列に観測された回数と共に示す．

```
  1 sys_CHDIR
  1 sys_SETSID
  1 sys_WAIT4
  2 sys_EXIT
  2 sys_WRITE
  3 sys_PRCTL
  5 sys_FORK
  6 sys_GETPID
  6 sys_IOCTL
  6 sys_TIME
 12 sys_CLOSE
 12 sys_NANOSLEEP
 13 sys_RT_SIGACTION
 24 sys_RT_SIGPROCMASK
 26 sys_FCNTL
 31 sys_SOCKETCALL
```

システムコールの観測時に確認できた挙動には，同じ処理をひたすら繰り返していることが挙げられる．これはマルウェアがネットワーク通信を行うために実行したシステムコールが失敗したことに起因する．プログラム上の処理ではシステムコールの呼び出し後に戻り値に基づいて接続の可否を判定する．従って，接続先ホストが存在しない等の理由で接続に失敗した場合，以降の処理は継続されない．図 1 は検体の逆アセンブル結果[2] であるが，検体は connect() の成功の可否に基づいて以降の処理を継続するかを判断している．

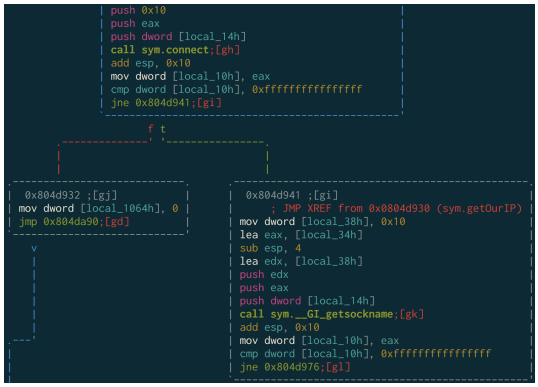

図 1　ネットワーク接続を試みる箇所

2.2 システムコールの戻り値の書き換えによる処理の継続

システムコールの観測により，非インターネット接続状態だとマルウェアの処理はループに陥る．これは通信に用いるシステムコールの connect() による接続が失敗したために戻り値が接続成功を示す値である 0 を返さなかったことによる．マルウェアが実行においてループ処理の状態に陥るのは，マルウェアが処理に利用するデータを取得できていないからである．そこで，システムコール呼び出しをフックし，処理を継続するために必要となる戻り値を受け渡すことでループ状態を回避できる．以下は Triton で実装した計装処理のコードの抜粋であるが，connect 関数の実行後に戻り値が格納される eax レジスタに 0 をセットすることで connect 関数が成功したと判断させる使用例で

† 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

ある．

```
def cb_syscall_exit(threadId, std):
    global current_Syscall
    if current_Syscall == "CONNECT":
        print "Detect CONNECT"
        setCurrentRegisterValue(
            getTritonContext().registers
            .eax, 0)

    if current_Syscall == "RECV":
        print "Detect RECV"
        setCurrentRegisterValue(
            getTritonContext().registers
            .eax, 1)
```

3. その他アプローチ

シンボリック実行を用いて，プログラムの実行時に特定の関数への到達に必要な入力値 (標準入力，ネットワーク入力) を解析する手法も存在する．これはマルウェアが C&C サーバから受信したデータを処理する機能の解析における活用が期待できる．シンボリック実行を利用するためのツールには angr[3] がある．検体を静的解析した結果 (図 2)，データ受信を担当すると思われる recvLine 関数と，C&C サーバから受信したデータを処理すると思われる processCmd 関数の呼び出しを発見した．そこで簡易的な使用例だが，angr[3] を用いて recvLine から processCmd へ到達するプログラム実行上のパスが存在するかを調査した．結果は実行後に 9 時間かけても終了せず期待したパスを発見できなかった．原因の 1 つに探索する対象の実行パスが膨大なことによるパス爆発 (Path explosion) が考えられる．パス爆発はシンボリック実行が抱える制限の 1 つである．この試みは失敗したが，これを解決するには実行ファイル内のコードの探索範囲を絞っていくことが考えられる．例えば目標となるコードブロックに到達する可能性の低いコードブロックは探索から除外する等の手法 (angr ではパス探索において回避したい命令アドレスを選択できる) が挙げられる．

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import angr
import sys
import claripy

START_ADDR = 0x804e249
AVOID_ADDRS = []
FIND_ADDRS = [0x0804e20d]
target = "
    fbb94cf7d98341e13dc1f6468f6e4e6b"
proj = angr.Project(target, load_options
    ={"auto_load_libs":False})
init_state = proj.factory.blank_state(
    addr=START_ADDR)
strstr = angr.SIM_PROCEDURES['libc']['
    strstr']
proj.hook_symbol('strstr', strstr())
@proj.hook(0x804e25f, length=5)
def my_hook(state):
```

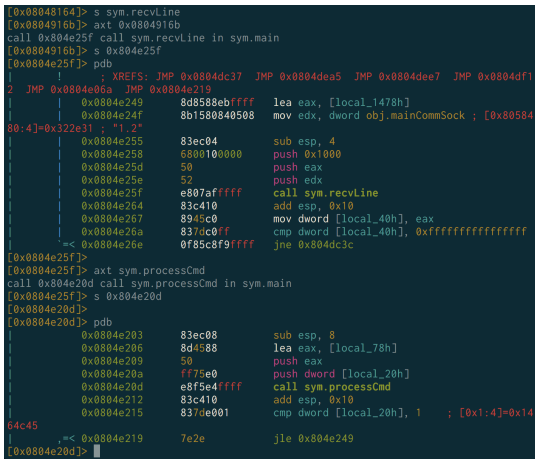

図 2　静的解析による C&C 処理の関数の位置の推定

```
    state.regs.eax = 1
sm = proj.factory.simulation_manager(
    init_state)
expl = sm.explore(find=FIND_ADDRS, avoid
    =AVOID_ADDRS, num_find=1)
print(expl)
s = sm.found[0]
print(s)
```

4. 結　　論

動的解析はマルウェア実行時においてシステム上での処理を観測するため解析結果から多くの情報が得られる．しかし動的解析の際には，接続先のサーバの存在やサーバから受け取る指令データといった諸々の条件を満たす必要がある．本発表では DBI を用いた環境チェックの回避方法や，実行中の処理に必要な入力値の発見におけるシンボリック実行の制限について実証した．解析の自動化に向けてはまだ課題が多く，静的解析で取得した関数名等から着目するコード箇所を絞り込む等，単一の解析手法だけでなく他の手法の利点を組み合わせる必要性が感じられる．

5. 謝　　辞

本実験を進めるにあたって横浜国立大学から IoT マルウェア検体[4] の提供に御協力を頂いたことに感謝します．

参　考　文　献

1) Triton - Dynamic Binary Analysis Framework https://triton.quarkslab.com/
2) Radare2 https://rada.re/r/
3) angr https://angr.io/
4) Minn, Yin Pa, et al. “IoTPOT: Analysing the rise of IoT compromises.” 9th USENIX Workshop on Offensive Technologies (WOOT). USENIX Association. 2015.