卒業論文

ハニーポットによる不正ファイルの入手と分析

2024 年度

拓殖大学工学部情報工学科

吉村　直将

指導教員 教授　蓑原 隆

助手　田島 信行

目次

図目次

第 1 章

はじめに

近年，サイバー攻撃の発生件数が年々増加してきており，その攻撃手法も多様化している．多様化した新しい攻撃に対処するためには攻撃手法の分析が必要である．攻撃手法の分析のために, 攻撃者を誘き寄せ，不正アクセスを受けるハニーポットを用いて攻撃者の情報を収集する方法がある．例えば過去の研究 [1] では，ハニーポットを利用して，ログイン試行時に使われる ID やパスワード，ログイン後に攻撃者から送られるシェルコマンド等の情報を収集する方法が提案されている

この研究では，Iot 機器への一定期間内で観測した攻撃データ，時期や頻度から将来的に攻撃に使用される ID やパスワードやシェルコマンドの解析から攻撃対象とされている Iot 機器の推測を立てることが示された．

また，ハニーポットは，攻撃を受け，攻撃内容を記録し，その攻撃手法を分析することで，攻撃への対策を強化することやデータ収集方法を改良することにもつながる．

本研究では，より具体的な攻撃者の攻撃手法の情報を得るため，攻撃者がログイン成功後に行う攻撃に着目し，ハニーポットを用いて，収集したコマンドから入手できるファイルの情報を収集し，ファイルの内容を解析するシステムを構築する．また，攻撃者から送信される攻撃コマンドを収集し，調査を行うことで，送信される頻度が多い攻撃コマンドの種類や攻撃対象とされている機器の種類の推測を立てる．

そして，攻撃の分析を行い，最新の攻撃内容について警告を発することを目的とし，セキュリティ対策を促すことに繋げる．

以下，本論文は次の構成で説明を行う．まず，２章で攻撃収集分析システムについて述べ，３章で研究に使用するハニーポットの詳細について述べる．ここでは，実施にハニーポットを運用して収集したデータについて，詳しく説明する．４章ではハニーポットに送られたコマンドを解析して不正ファイルを作成しようとする攻撃に対して，データをダウンロードするプログラムについて説明する．５章ではダウンロードした不正ファイルの解析について，手作業の結果と自動化について説明する．最後に６章で本研究のまとめを述べる．

第 2 章

攻撃収集分析システム

攻撃者がダウンロードさせようとしてくる不正なソフトウェアの解析を実現する為のシステムの構成を図 2.1 に示す.

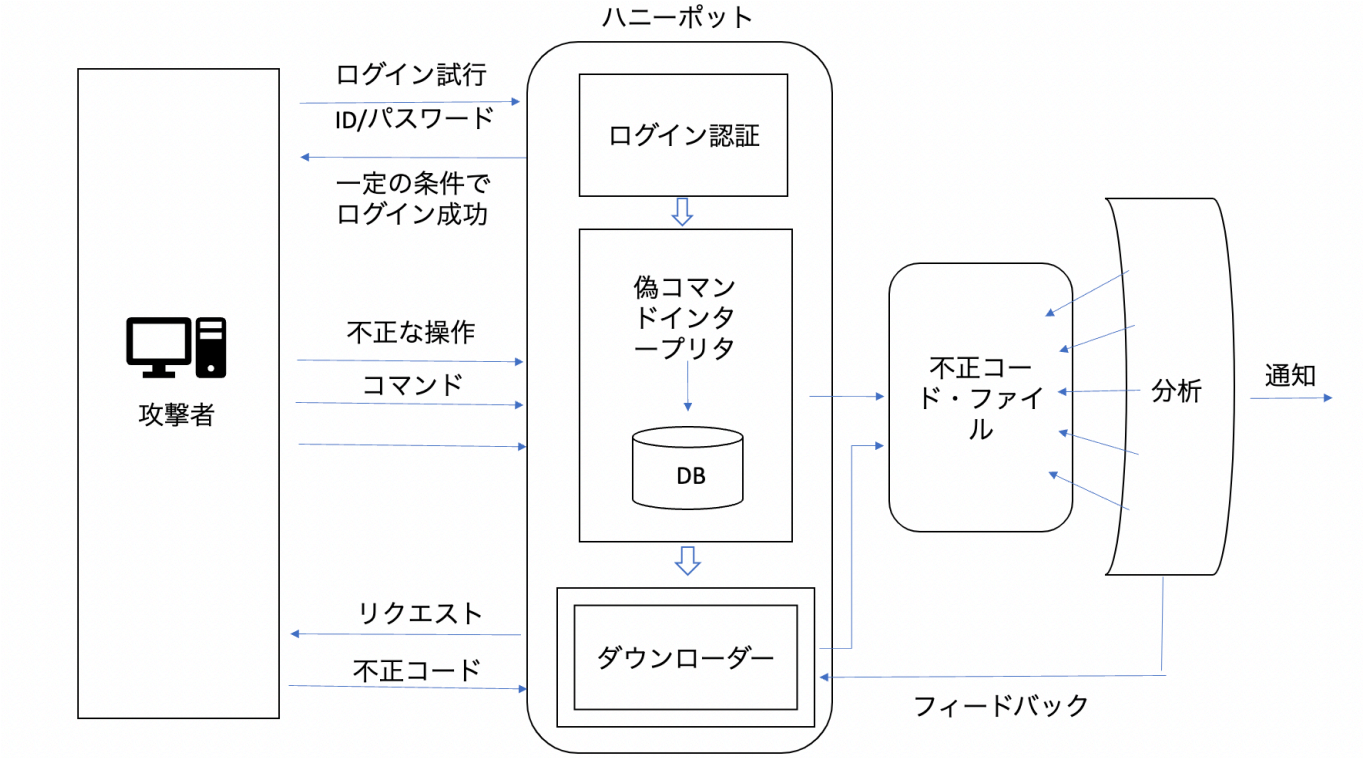

図 2.1　システムの構成

ハニーポットは，ログイン認証部で，攻撃者からの何度かのログイン試行として送られてくる ID/パスワードを受け，一定の条件で，攻撃者にログイン成功したと思わせる．その後，偽コマンドインタープリタ部で攻撃者の不正な操作に対して，コマンドインタープリタの様な返答を見せる形で対応し，不正な操作のコマンドをデータベース DB に収集する．収集したコマンドから，ダウンローダー部で攻撃者が不正なサイトからダウンロードさせようとする不正なファイルを安全な場所で，対策を取りながら入手する．また，コマンドの中には，ハニーポット内に不正ファイルを直接作成させようと試みるものもあり，安全にファイルを作成し，収集を行う．収集した不正ファイルのコードから，どの様な不正ファイルかを分析し警告を発する．また，その情報からダウンローダーに生かせるものをフィードバックする．

本研究では，ハニーポットとして DShield (Distributed Intrusion Detection System)[2] と呼ばれるグローバルなセキュリティコミュニティによって構築された分散型侵入検知システムを使用する．Dshield は，世界中のネットワーク上で発生するセキュリティイベントのデータを収集し，分析することでセキュリティの脅威情報を提供する．これまで我々の研究室では主にログイン試行時の Id やパスワードを収集することで攻撃頻度の時系列解析を行うために Dshield ハニーポットを利用している [3]．が，本研究では図 2.1 に示すように，Dshield の cowrie[4] のログイン認証部，コマンドインタープリタ

部に必要な機能を追加する，また，コマンドからファイル又は，URL などを取集するダウンローダー部，不正ファイルの分析部は，新しくプログラムを作成する．

第 3 章

Dshiled の環境構築と運用

本章では，研究に使用するハニーポットの詳細について述べる．また，実施にハニーポットを運用して収集したデータについて，詳しく説明する．

3.1　ハニーポットの環境構築

Raspberry Pi に Dshield をインストールし，パスワードや接続する無線 LAN の設定を行なった．Rassberry Pi のファイアウォールの設定から SSH を有効にする事で，外部からの接続を cowrie が対応するように設定した．また，研究室内のネットワークからの接続は攻撃と見さないように設定した．

具体的に行った設定作業を以下に示す．

- まず初めに，本研究で使用する Raspberry Pi として，Raspberry PI 400 を用意し，ディスプレイに繋げて，無線 LAN やパスワードの再構成を行った．
  Raspberry PI 400 は，見た目はキーボード型で，キーボードに Rasberry PI の機能が内蔵されており，モニターやマウスや SD カードなどを接続することでデスクトップとして起動できる．キーボードは 83 キーの日本語配列となっている．また，raspberryPi 4 からさらに高性能になっている新リビジョン SoC1.8GHz や 4GB の RAM を搭載している．
- 次に，ディスプレイの設定から Raspberry Pi の設定，その中のインターフェースの SSH 有効を選択することで，SSH を通って外部からの接続を可能にした．
- さらに，Raspbery PI のソフトウェアのアップデートを行った
  ```
  sudo apt -uy dist-upgrade
  sudo apt update
  ```
- そして，Raspberry Pi に Dshield をインストールするために，git をインストールした．
  ```
  sudo apt -y install git
  ```
- ハニーポット Dshield のソフトを GitHub で公開されているリポジトリから clone した．
  ```
  git clone https://github.com/DShield-ISC/dshield.git
  ```
- クローンしたファイルの中にインストールスクリプトがを用意されているので，それを以下のようにして実行した．
  ```
  cd dshield/bin
  sudo ./install.sh
  ```
- インストールに並行して，デバックログファイルを確認する
  ```
  sudo tail -f LOGFILE
  ```
- エラーが起きず，スクリプトが正常に完了した後に，デバイスを再起動する．
  ```
  sudo reboot
  ```

次に，研究室内のネットワークをファイヤウォールで check しないように設定した．ハニーポットの設定をする際には，

専用のコマンドとして ssh [パスワード] pi@[ハニーポットに繋がっている IP アドレス] で接続することができる．

3.2　ハニーポットの試用

攻撃者からコマンドを収集するために Dshield のプログラムを調査し，コマンドを収集できているのか確認した．

調査の結果，Dshield は Cowrie[4] を使用して攻撃コマンドを取集し，/srv/cowrie/var/log/cowrie の場所にファイル名が cowrie.log や cowrie.json に日付が加わった形で保存していることが分かった．また，ログイン試行に対応しているプログラムが/src/cowrie/core/の場所にある auth.py であることを突き止め，解読したところ，Dshield は外部からの攻撃者からのログイン試行を 1 回以上のランダム回数行うと，ログイン可能とするように設定されていることが分かった．

3.3　ハニーポットの運用

実際にハニーポットを運用し 5/19 から 5/30 の期間中にコマンドを取集した．

3.3.1　調査プログラムの作成

収集した攻撃データのファイル形式は Json 形式であり，攻撃コマンド情報だけではなく色々な情報を確認することができたことから，ファイルデータを扱いやすく，且つ分かりやすくするためのプログラムを複数作成した．

初めに，コマンド引数でした指定した一つの json ファイルから一行ずつ Json データを辞書形式に変更するプログラムとして readJson.py を作成した．また，攻撃者が変わった際に分かりやすいように，ip アドレスが変わったらその都度表示するようにした．readJson.py を図 3.1 に示す．

```
1  with open(sys.argv[1]) as f: # コマンド引数のファイルを開く
2      ip = None
3      cmd = 'wget'
4      for line in f: # ファイルから一行ずつ読み込む
5          dic = json.loads(line) # 一行のJSON データを辞書形式に変換する
6          if dic['eventid'] == 'cowrie.command.input':
7
8         #if dic['src_ip'] != ip:
9          # ip = dic['src_ip']
10         # print('[src_ip] =', dic['src_ip'])
11        #print(dic['input'])
12
13            if cmd in dic['input']:
14                 if dic['src_ip'] != ip:
15                    ip = dic['src_ip']
16                    print('[src_ip] =', dic['src_ip'])
17                print(dic['input'])
```

図 3.1　readJson.py

cowrie の Json データの方式は，キーと値の形になっており，5 行目で一行ずつ Json データを辞書形式に変換する．6 行目でキー ['eventid'] の値が cowrie.command.input であることを確認し，攻撃者に打たれたコマンド情報であるかの判定を行う．そして，キー ['input'] の値がコマンドの情報となっている．また，3 行目で変数 cmd を用意し，今回は，wget を入れ，13 行目で，コマンド情報に wget が入っているかの判定を行うことで，求める対象の攻撃コマンドを狭めて確認できるようになっている．その際，攻撃者が変わったことが分かりやすいように，2 行目で事前に変数 ip を用意しておき，14,15 行目で，ip アドレスが変わるごとに更新する．

次に複数日の json ファイルを一括で処理するプログラムを shell スクリプトで作成した．作成したプログラム forList.sh

は，コマンドで指定したファイル (foo) に書かれているファイルを対象に readJson.py を実行する．

forLsit.sh を図 3.2 に，foo ファイルを図 3.3 に示す．

```
#!/bin/bash
for i in `cat $1`; do
    python3 readJson.py $i
    #python3 searchJson.py $i
done
```

図 3.2　forList.sh

```
cowrie/cowrie.json.2023-05-19
cowrie/cowrie.json.2023-05-20
cowrie/cowrie.json.2023-05-21
cowrie/cowrie.json.2023-05-22
cowrie/cowrie.json.2023-05-23
cowrie/cowrie.json.2023-05-24
cowrie/cowrie.json.2023-05-25
cowrie/cowrie.json.2023-05-26
cowrie/cowrie.json.2023-05-27
cowrie/cowrie.json.2023-05-28
cowrie/cowrie.json.2023-05-29
cowrie/cowrie.json.2023-05-30
cowrie/cowrie.json.2023-05-31
```

図 3.3　foo

3.3.2　収集された攻撃コマンド

forLsit.sh を foo ファイル対象として実行した結果を図 3.4 に示す．

```
[src_ip] = 180.116.50.76
tftp; wget; /bin/busybox BCJAY
[src_ip] = 59.178.216.135
tftp; wget; /bin/busybox YWWGW
[src_ip] = 180.116.50.76
tftp; wget; /bin/busybox PEZJT
[src_ip] = 185.224.128.121
lscpu | grep "CPU(s):                " && echo -e "cKrEWeqRzXwt\ncKrEWeqRzXwt" | passwd && cd /tmp; wget http
://84.54.50.198/pedalcheta/cutie.x86_64; curl -s -O http://84.54.50.198/pedalcheta/cutie.x86_64; chmod 777 cu
tie.x86_64; ./cutie.x86_64 x86h
[src_ip] = 113.161.4.96
tftp; wget; /bin/busybox BSMKI
```

図 3.4　forList の実行結果

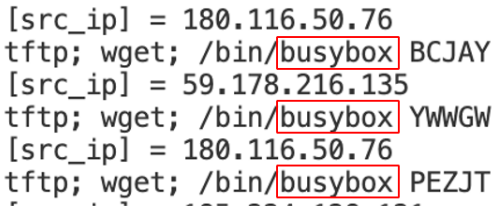

図 3.5　busybox の攻撃コマンド

収集したコマンドの中には, 特定のコマンドが多く発見された．例えば組み込み Linux で複数のコマンドをまとめるために使われる busyBox が図 3.5 のように含まれていて，busyBox の後にオプションだと思われる 5 桁のランダムな英字が続くものが含まれた攻撃コマンドは，コマンドの中でも特に多かった．そのため，この期間中に多くの攻撃が組み込み Linux の機器を対象としていることが分かった．

また，色々な ip アドレスから同じパターンで送られてくる攻撃コマンド図 3.6 が多く見られた．この攻撃コマンドの内容

```
[src_ip] = 220.134.216.159
enable
system
shell
sh
cat /proc/mounts; /bin/busybox FBYSL
cd /dev/shm; cat .s || cp /bin/echo .s; /bin/busybox FBYSL
tftp; wget; /bin/busybox FBYSL
dd bs=52 count=1 if=.s || cat .s || while read i; do echo $i; done < .s
/bin/busybox FBYSL
rm .s; exit
```

図 3.6　多くみられた攻撃コマンドのパターン

から攻撃者の意図は，次のように考えられる．

- 初めに enable や system コマンドで，管理者権限を上げ，より重要なコマンドを使用可能にする．
- 次に，shell や sh でシェルを起動し，コマンド入力で操作できる環境を作る．というように，同じ内容の攻撃コマンドを複数試し，成功するかどうかで，接続先がどのような OS かなどの情報を絞り込んでゆく．
- そして，cat コマンドを用いて実行中のファイル情報や OS 情報を持つ proc/mount のファイルの中身を確認する．
- このパターンの攻撃コマンドは，busybox の後の英文字がランダムで他の部分は変わらない．
- `cd /dev/shm;cat .s` で，データを共有するファイルに移動し，.s ファイルの中身を見る．前の動作のコマンドが失敗したら `cp /bin/echo .s` で，/bin/echo を .s という名前でコピーする．
- もし，/bin/echo が busybox で実装されていれば，.s は busybox のコピーということになる．
- tftp でファイルの転送，wget でファイルのダウンロードを行う．
- 次に `dd bs=52 count=1 if=.s` は，dd でデータを変換コピーする．bs=52 で，ファイル入力時のブロックサイズを 52 とし，count=1 で，コピーするブロックサイズを 1 としているので，.s ファイルの先頭 52 バイトを出力することになる．
- dd コマンドが失敗したら，`cat .s` で，.s ファイルをの中身を見る．
- さらに，`while read i;` で 1 行ずつ読み込んで，`do echo \$i;` で 1 行ずつ echo で表示をし，`done < .s` のようにして，.s ファイルの内容を表示することを試みている．
- この 3 つ全てが.s ファイルを見る動作で，色々な方法を試し，どれか成功すればよいという攻撃者の考えが見られる．
- 最後に `rm .s;exit` で.s ファイルを排除し，exit で抜ける．

収集したコマンド中には，不正なファイルをダウンロードさせるため wget コマンドを用いているもの図 3.7 や，不正なファイルをハニーポット内に作成するために，echo コマンドを用いている攻撃コマンド図 3.8 が多かった．

```
[src_ip] = 185.224.128.121
lscpu | grep "CPU(s):                " && echo -e "cKrEWeqRzXwt\ncKrEWeqRzXwt" | passwd && cd /tmp; wget http
://84.54.50.198/pedalcheta/cutie.x86_64; curl -s -O http://84.54.50.198/pedalcheta/cutie.x86_64; chmod 777 cu
```

図 3.7　wget の攻撃コマンド

```
[src_ip] = 95.214.27.202
cd ~ && rm -rf .ssh && mkdir .ssh && echo "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQC/yU0
iqklqw6etPlUon4mZzxslFWq8G8sRyluQMD3i8tpQWT2cX/mwGgSRCz7HMLyxt87olYIPemTIRBiyqk8SLD3ij
QpfZwQ9vsHc47hdTBfj89FeHJGGm1KpWg8lrXeMW+5jIXTFmEFhbJ18wc25Dcds4QCM0DvZGr/Pg4+kqJ0gLyq
YmB2fdNzBcU05QhhWW6tSuYcXcyAz8Cp73JmN6TcPuVqHeFYDg05KweYqTqThFFHbdxdqqrWy6fNt8q/cgI30N
```

図 3.8　echo の攻撃コマンド

第 4 章

ダウンローダー部の作成

ダウンローダー部のシステムプログラムは，主に 3 つの動作を行う．

(1) 不正ファイルデータのダウンロードコマンドを模倣して不正データを入手する
(2) 不正データを安全に扱うためにヘッダ情報を追加する
(3) データを保存するファイル作成し，ヘッダ情報と不正ファイルデータを書き出す．

(1) について，Dshield が Cowrie によって攻撃コマンドを模倣している部分を調査した．まず初めに，攻撃者に決まった内容の応答を返すコマンドの応答内容が図 5.1 のように txtcmd というディレクトリの下のファイルに置かれていることが分かった．

```
% tree txtcmds
txtcmds
├── bin
│   ├── df
│   ├── dmesg
│   ├── enable
│   ├── mount
│   ├── stty
│   ├── sync
│   └── ulimit
└── usr
    ├── bin
    │   ├── clear
    │   ├── emacs
    │   ├── getconf
    │   ├── killall
    │   ├── locate
    │   ├── lscpu
    │   ├── make
    │   ├── nano
    │   ├── nproc
    │   ├── pico
    │   ├── pkill
    │   ├── top
    │   ├── vi
    │   └── vim
    └── sbin
        └── vipw
```

図 4.1　固定的な応答内容のファイル

次に，このファイルを使っているソースプログラムを次のコマンドで検索し，/srv/cowrie/shell/protocol.py の getCommand 関数で使われていることが分かった．

```
find /srv/cowrie/ -exec grep, ’ textcmds’ {}\; -and -print 2>/dev/null
```

getCommand 関数は/srv/cowrie/shell/honeypot.py で使われており，この honeypot.py の LineReceived 関数で， 攻

撃者が入力された行の処理を行うときに，runCommand 関数を呼び出して protocol.getCommand() コマンドを実行していることがわかった．

さらに，応答が固定的でないコマンドは，srv/cowrie/src/cowrie/commands の下に Python プログラムとして実現されていることが分かった．例えば，不正ファイルデータのダウンロードに使われている wget の動作は，wget.py 中で行われる．

```
123         self.deferred = self.download(self.url, self.outfile)
124          #URL と out ファイルを引数に download 関数で不正サイト情報 defferd を取得
125         if self.deferred:
126             self.deferred.addCallback(self.success) # 情報を得られたらsuccess メソッドへ
127             self.deferred.addErrback(self.error, self.url) # 情報を上手く得られなかったら
                    error メソッドへ
128         else:
129             self.exit()
```

図 4.2　wget コマンドのフリをするプログラム (wget.py) の一部

wget.py の一部を図 4.2 に示す．123 行目で `self.download(self.url, self.outfile)` に引数として URL とファイル名を渡して，ダウンロードを実行する．

download 関数では，twisted ライブラリ [5] を使用して，HTTP 通信を開始する．ネットワーク通信に時間が掛かるので，125 行目から 127 行目で，通信終了時に呼び出される関数を登録している．通信が成功したら，`self.deferredaddCallback(self.success)` で登録した success 関数が呼び出される．

(2) と (3) の処理については wget.py の success 関数にコードを追加して実現した．追加したコードを図 4.3 に，wget.py 全体のリストを付録 A.1 に示す．

(2) において，ダウンロードしたデータをそのままファイルとして保存することを避け，不正データのファイルを誤って動作させてしまった場合でも問題が起きないようにする安全対策のために，ヘッダ情報を不正データに追加する．

具体的なヘッダ情報としては，ダウンロードに使用した URL の前に 4 桁の URL 文字数を追加したものとし，URL の文字数は，4 桁の数字を追加したものとする．

(3) については，Dshield ハニーポットが攻撃者に見せているファイルシステムの外の領域として外付け HDD(/HD/malwares/tmp/) に作成する，そしてファイル名は，図 4.3 の 234 行目のように，MW の後に攻撃があった日時を入れたものとする．

図の 236 行目から 239 行目が実際にファイルを書き込む部分である．download 関数がダウンロードしたデータは変数に記録されているので，ヘッダの後にその内容を書き出している．

実際に研究室で運用しているハニーポットに組み込んで確認したところ，攻撃があったときにデータが記録されていないことが判明した．原因は DShield が 1 日に 1 回,18 時 28 分に，配布元を確認し，システムのアップデートがあった時，自動的に更新しているためであった．自動更新が行われると，修正した wget.py が上書きされてしまい，追加した処理が行われていなかった．

そこでハニーポットの/etc/cron.d/dshield に変更を加え，アップデートが終了したあとで，wget.py へ周期的に変更を加えたもので上書きをするように設定し，正しく動作していることを確認した．

第 5 章

不正ファイルの分析

本章では，ダウンロードした不正ファイルの解析について，手作業の結果と自動化について説明する．

5.1　手作業によるファイルの分析

不正ファイルの分析として実際の不正ファイルからの情報取得を行った．具体的には，直接不正ファイルを作成させようとしてくる攻撃の 1 つとして，複数の echo コマンド を次のように送るものについて，実際にファイルを作成して調査した．

```
echo -ne "\x7f\x45.." > niggabox
```

以下，作成したファイルを対象データと呼び． その先頭部分を図 5.1 に示す．

```
1  0000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
2  0000010 02 00 28 00 01 00 00 00 2c 83 00 00 34 00 00 00
3  0000020 78 04 00 00 02 00 00 04 34 00 20 00 04 00 28 00
4  0000030 0a 00 07 00 01 00 00 00 00 00 00 00 00 80 00 00
5  0000040 00 80 00 00 d0 03 00 00 d0 03 00 00 05 00 00 00
6  0000050 00 80 00 00 01 00 00 00 d0 03 00 00 d0 03 01 00
7  0000060 d0 03 01 00 10 00 00 00 10 00 00 00 06 00 00 00
8  0000070 00 80 00 00 07 00 00 00 d0 03 00 00 d0 03 01 00
9  0000080 d0 03 01 00 00 00 00 00 08 00 00 00 04 00 00 00
10 0000090 04 00 00 00 51 e5 74 64 00 00 00 00 00 00 00 00
```

図 5.1　対象データの先頭部分

まず，対象データは 2,720 バイトのバイナリーデータであることがわかった．バイナリーデータのファイルは，最初の数バイトがファイル形式を示している場合が多い．対象データについて先頭部分を調べたところ，最初の 4 バイトが，“7f 45 4c 46” であり，ELF 形式 [6] のファイルであると判別した．

次に ELF 形式のヘッダ情報の解析を行う．ELF 形式のファイル構造では，

- 1 から 4 バイト目がマジックナンバーとして”7f 45 4c 46”に固定されている
- 5 バイト目がクラスとして，32 ビットオブジェクト (1) か，64 ビットオブジェクト (2) かを示している
- 6 バイト目がデータの符号化として，リトルエンディアン方式 (1) かビックエンディアン方式 (2) かを示している
- 18 バイト目が対象としているアーキテクチャの種類 (28 は ARM) を示している

今回の対象データは，32 ビットのリトルエンディアン形式で，ARM アーキテクチャ用であることがわかる．

そこで，次のコマンドで逆アセンブリを行い内容を確認したところ，

`objdump --print-imm-hex` –disassemble -s niggabox!

マルウェアの Mirai のソースコード [7] として公開されているコードの一部に酷似していることがわかった．

さらに，リンク先から別のファイルをダウンロードする機能を持っていることが分かり．具体的にこの不正ファイルでは，リンク先はアムステルダムで，別のファイルは jklarm7 というファイルであることも分かった．

5.2　ELF 形式のファイルから情報取得の自動化

ELF 形式のファイルについて，自動で情報を解析するためのプログラムを作成した．

ELF 形式は図 5.2 に示すように, ELF ヘッダ (ELF Header) につづくデータ部分になっており，データ部分は，プログラムヘッダテーブル (Program header table) または，セクションヘッダテーブル (Section header table) の情報にしたがって，実行時にメモリ上に展開される.text，.rodata などの複数セクションで構成されている．

プログラムヘッダテーブルとセクションヘッダテーブルのサイズやオフセットの情報は，ELF ヘッダから取得できる．また，各セクションの境界はセクションヘッダの情報で切り分けられる．セクションヘッダテーブルは，エントリという項目で分かれており，各エントリは一つ一つのセクションの情報が表 5.1 に示すように書かれている．

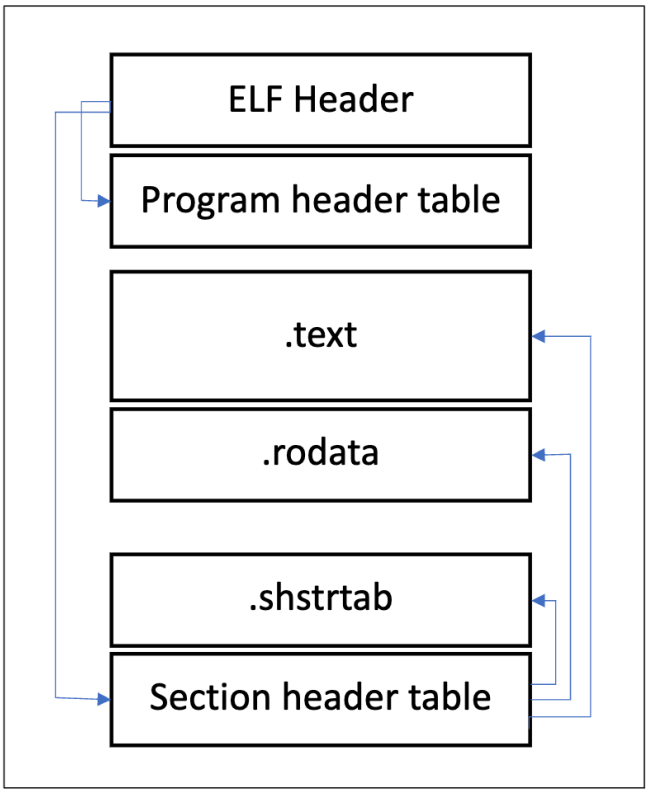

図 5.2　ELF 形式のファイル構造

表 5.1　セクションヘッダのエントリー情報 (一部)

エントリ	名前	意味
0	sh_name	セクション名が格納された shsrttbl のインデックス
4	sh_offset	ELF ファイル中のデータのオフセット
5	sh_size	ELF ファイル中のデータのサイズ

自動情報取得として，プログラムに埋め込まれた文字列などのリテラルが格納されている.rodata から情報を取得する処理をするプログラム elf.py を図 5.3, 5.4, 5.5 に示すように実装した．

elf.py では，図 5.3 に示すように，初めに ELF 形式のファイルであるか判別し，次に，ファイルの基本情報をファイルのヘッダテーブルから取得する．

その基本情報を元に，図 5.4 に示すように，データを struct モジュールの unpack を使用して，データ方式にのとった方法で変換していく，例えば 26 行目の，`e1 = struct.unpack("<lll",data[24:24+12])` では，ファイルデータの 24 バイ

```
1  import struct
2  f = open("niggabox","rb")
3  data = f.read()
4  if data[0:4] == b'\x7fELF':
5      print("ELF")
6      match data[4]:
7          case 1:
8              print("32bit")
9          case 2:
10             print("64bit")
11     match data[5]:
12         case 1:
13             print("little endian")
14         case 2:
15             print("big endian")
16     match data[18]:
17         case 0x07:
18             print("Intel 80860")
19         case 0x13:
20             print("Intel 80960")
21         case 0x28:
22             print("Arm")
```

図 5.3　elf.py (1/3)

ト目から 12 バイト分のデータを 4 バイトずつリトルエンディアン方式で，e1 として取得し，エントリーポイントのアドレス，および，プログラムヘッダーテーブルやセクションヘッダテーブルのオフセットやサイズを取得する．

同様に，ファイルデータの 40 バイト目から 2 バイトずつ 6 個のデータを，e2 として取得し，各テーブルのエントリサイズと数を取得する．さらに，セクションヘッダに記述された各セクションの名前が格納されている 「.shstrtab」というセクションのインデックスの情報 shstrndx を取得する．ELF 形式では各セクションが記録される順番は規定されていないが，.shstrtab セクションには，各セクション名のテーブルが格納されており，名前を使ってセクションの判別を行うことができる．

図 5.5 では，まず，セクションヘッダテーブルのセクションごとのオフセットやサイズ情報を unpack し，リスト (sehd) に追加している．

次に，セクションヘッダリスト shed の中の.shstrtab のセクションを shstrndx が指定しいるので，53 行目に示すように，そのエントリのオフセット (4 番目) とサイズ (5 番目) を使って，.shstrtab のデータを shstr として取得する．

さらに，表 5.1 に示したように，セクションヘッダテーブルのエントリの最初の項目が，shstrtab の'0' で終端されたセクション名のオフセット情報を持つことから，58 行目から 62 行目のようにループを使ってセクション名 sname を取得する．

各セクションの名前を取得するときに 64，65 行目でセクション名が.rodata かを判定し，.rodata のエントリのインデックスを rden として記録する．

最後に，67，68 行目で，.rodata のセクション内容を表示している．

手作業での分析に使用した対象データ (niggabox) について，作成したプログラムを実行し，図 5.6 のようにダウンロードしようとするファイル名 (jklarm7) などの情報が取得できることを確認した．

```
# エントリポイント,プログラムヘッダテーブル,セクションヘッダテーブルの取得
 if data[4] == 1: #16bit の場合
    e1 = struct.unpack("<lll",data[24:24+12])
    entrypoint = e1[0]
    print("entrypoint = " + hex(entrypoint)) #エントリーポイントのメモリアドレス
    phoff = e1[1] #str(data[28] + data[29] + data[30] + data[31])
    print("phoff = " + hex(phoff)) #プログラムヘッダテーブルの最初のポイント
    shoff = e1[2]#str(data[32] + data[33] + data[34] + data[35])
    print("shoff = " + hex(shoff)) #セクションヘッダテーブルの最初のポイント
    e2 = struct.unpack("<6H",data[40:40+12]) #2byte ずつ
    ehsize = e2[0] #str(data[40] + data[41])
    print("ehsize = " + hex(ehsize)) #このヘッダーのサイズ
    phetsize = e2[1] #プログラムヘッダテーブルのエントリーサイズ
    print("phetsize = " + hex(phetsize))
    phnum = e2[2] #プログラムヘッダテーブルのエントリー数
    print("phnum = " + hex(phnum))
    shensize = e2[3] #セクションヘッダテーブルのエントリーサイズ
    print("shensize = " + hex(shensize))
    shennum = e2[4] #セクションヘッダテーブルのエントリー数
    print("shennum = " + hex(shennum))
    shstrndx = e2[5] #セクションヘッダテーブル内でセクション名を持つエントリの位置
    print("shstrndx = " + hex(shstrndx))
```

図 5.4　elf.py (2/3)

```
    sehd = []
    for i in range(shennum):
        shptr = shoff + i * shensize #エントリのスタート位置
        sehd.append(struct.unpack("<10l",data[shptr:shptr+shensize]))
    print(sehd)
    print(sehd[shstrndx][4]) #.strtab のオフセット
    shstr = data[sehd[shstrndx][4]:sehd[shstrndx][4]+sehd[shstrndx][5]]

    rden = 0
    for i in range(shennum):
        sname = ''
        for ch in shstr[sehd[i][0]:]:
            if ch == 0:
                break
            else:
                sname = sname + chr(ch)

        if sname == ".rodata":
            rden = i

    print(".rodata:" + str(rden))
    print(data[sehd[rden][4]:sehd[rden][4]+sehd[rden][5]])
```

図 5.5　elf.py (3/3)

図 5.6　ELF 形式データからの情報抽出結果

第 6 章

まとめ

本研究では，Dshield ハニーポットに，ダウンローダー部や解析を行うプログラム elf.py などの機能を追加することで，ダウンローダー部では攻撃コマンドが送り込もうとしている不正ファイルデータを取得し，elf.py では ELF 形式のファイルの解析を行うシステムの開発を行なった．

そして，我々の研究室で運用しているハニーポットに実装した，運用をしていきながら攻撃データを収集していった，収集した攻撃コマンドには，特定のパターンのコマンドが多く発見され，busybox というコマンドが多く使われており，取集期間である．5/19 から 5/30 期間中は，多くの攻撃が組み込み Linux を攻撃対象としていることが分かった．また，wget や echo のような攻撃コマンドが確認でき，その攻撃コマンドを利用し，ダウンローダー部で不正ファイルデータの取得を行った，そして，elf.py を用いた解析結果の表示を行えることを確認した．具体的な解析結果として，攻撃コマンド echo から取得した不正ファイルデータの一つが Mirai と呼ばれるマルウェアに酷似しており，リンク先であるアムステルダムから別のファイル jklarm7 をダウンロードする機能を持っていることが分かった．

本研究では，攻撃収集分析システムとして解析結果を管理者に通知する予定であったが，ハニーポットから通常のネットワークに通信を行う仕組みを安全に実現することがこんなんであったため，通知部分は未実装で残された課題となっている．

参考文献

[1] 中山楓, 鉄穎, 楊笛, 田宮和樹, 吉岡克成, 松本勉：IoT 機器への Telnet を用いたサイバー攻撃の分析, 情報処理学会論文誌, Vol. 58, No. 9, pp. 1399–1409 (2017).

[2] DShield Honeypot https://isc.sans.edu/tools/honeypot/ (accessed accessed 2024/6/14).

[3] 西田圭介：インターネット上のサイバー攻撃のハニーポットを用いた分析と可視化, 拓殖大学工学部情報工学科卒業論文 (2022).

[4] Welcome to Cowrie's documentation! https://cowrie.readthedocs.io/en/latest/index.html (accessed accessed 2024/5/31).

[5] Welcome to the Twisted documentation! https://docs.twisted.org/en/stable/index.html (accessed accessed 2024/07/28).

[6] elf - 実行可能リンクフォーマット (ELF) ファイルのフォーマット https://manpages.ubuntu.com/manpages/trusty/ja/man5/elf.5.html （参照参照 2024/6/14).

[7] Mirai BotNet https://github.com/jgamblin/Mirai-Source-Code (accessed accessed 2024/5/31).