

情報ネットワーク応用 演習レポート

学籍番号 1280391
細川 夏風

2026年2月11日

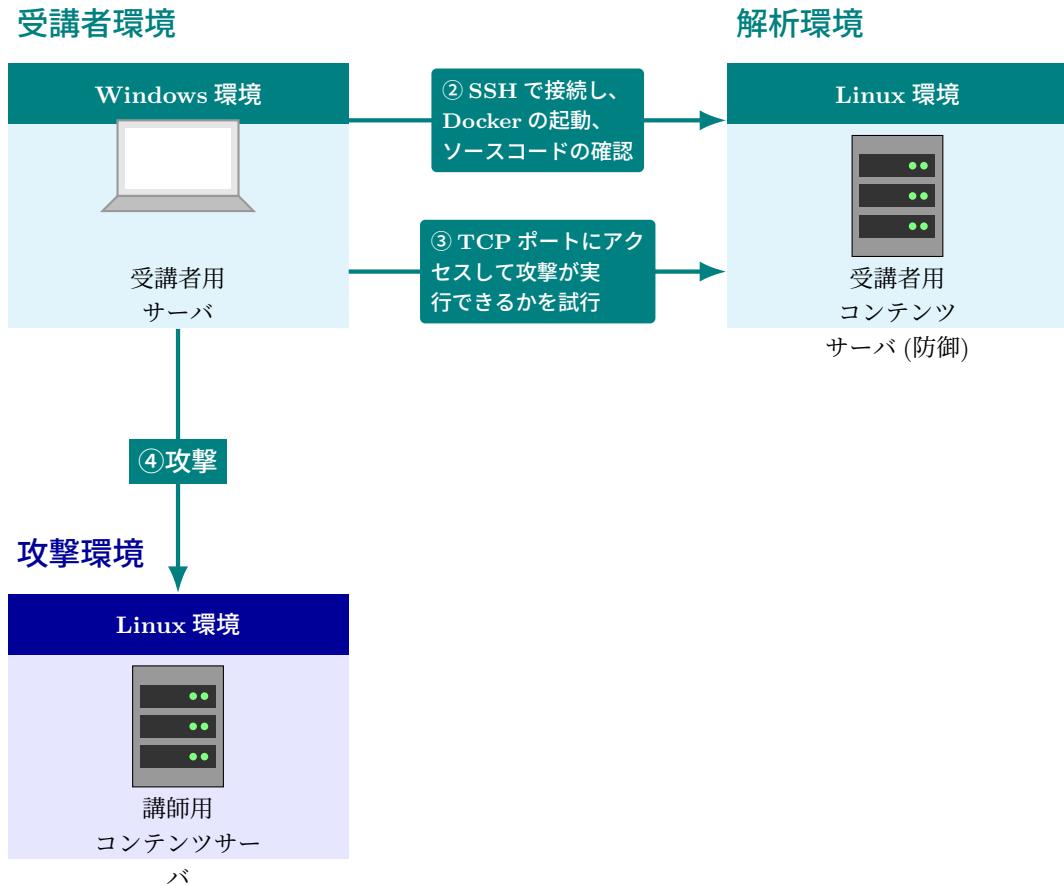
1 背景

この世の多くのサービスに C 言語が使用されている。その理由は非常に高い拡張性と高速性である。しかし、C 言語には多くの問題がある。それがメモリに関するものである。C 言語は境界値の検査をしない。すなわち、確保された領域に対してそれを超える操作をした際にそのメモリのを超えた先にアクセスできてしまう。これはバッファオーバーフローを引き起こす大きな問題であり、それに対処するために標準入力や標準出力に対して制限をつけたものが使用されるケースが一般的だ。しかし、それが行われていないケースも存在する。それらがどのような被害を引き起こすかについて今実習から議論を行う。

2 方法

今回は演習 1, 2, 3 について記している。しかし、演習 3 は演習 1 とほとんど同じであるため、記述を省くこととする。

環境



手順

2.1 演習 1

- (1). Docker を起動しコンテナ内に入ると

```
1 cat hello.c
```

を行い、ソースコードの解析を行う。

- (2). この時、C 言語の `gets()` が用いられていることとどこからも参照されていない `shell` というターミナルのコマンドを実行できる関数が用意されていることがわかる。
(3). 次に実行ファイルに逆アセンブルを行う。これによってどの関数やどのプログラムがメモリ上のどこで動作するかがわかる。

```
1 objdump -d hello
```

- (4). PwnTools という脆弱性をつく Python のオープンソースライブラリを用いて、攻撃を行う。
(5). またこの時、`remote` という python モジュールを用いる。これによって一時的に対象の IP アドレスの特定のポートに対して接続を行える。
(6). Python によって送信する特定のデータを作成する。この時、先ほど解析した結果から `return` が呼び出されている場所に飛ばす。理由は `return` 命令とは単にスタックポインタの先頭のアドレスにジャンプするだけの `jmp` 命令と大差ないからだ。
(7). これから `gets()` と `return` 命令には 72bytes 分の差がある。よってこれを適当な値で埋める。今回は A を 72 個で埋めている。
(8). その後に `shell` 関数のアドレスを入力すればいいのだが、この部分で 1 つ問題がある。x64 には制約があり、それは関数呼び出しの際は 16 の倍数に調整しなければならないという点だ。そのため、再度 `return` を呼び、8bytes 分追加してから `shell` 関数の呼び出しを行う。
(9). 最後に対策であるが、これは単純である。`fget()` を用いて入力に制限をかける。今回は変数 `username` 分という制限を設けた。

2.2 演習 2

- (1). Docker を起動してコンテナ内に入ると

```
1 cat shellcode.c
```

- (2). この時、ソースコード内に `(*void (*)())buf();` というプログラムがあることを確認できる。
(3). このプログラムは `void` 型関数へのポインタという意味であり、`void` を返す関数のポインタを持っている。正確には `buf` という領域に確保された `void` を返す関数へのポインタという意味

である。

- (4). よって入力された `void` を返す関数を実行してしまうのだ。
- (5). その機能を携えた Python ファイルを実行することにより、バッファオーバーフローが可能になる。

3 考察

バッファオーバーフローは非常によく用いられる攻撃手法である。理由は攻撃者の行える事の多さにある。多くの場合で自身の任意のプログラムを実行可能である。これによってより複雑な攻撃も可能になるのだ。過去の事例では SQL Slammer がある。これは Microsoft の SQL サーバーのバッファオーバーフロー脆弱性を利用したものであり、これはたった 376bytes ととても小さい容量で UDP パケットに収まるほどであったと言われている。感染開始からわずか 10 分で脆弱性のある世界中のサーバの約 90% に感染した [?].

境界チェックをしていなければそのほとんどはいくつかの適当な値で埋められたデータとともに自身が動かしたいプログラム、それに加えてそのプログラムのアドレスさえあればバッファオーバーフローが可能である。最初の適当な値で埋められたデータは `return` や `jmp` などメモリ上のどこかへ飛ばす処理までの穴埋めに用いられる。次のプログラムは単に実行したいプログラムそのものである。最後のそのプログラムへのアドレスはそのプログラムを書くだけでは意味がない。その関数やプログラムまでの道のりが必要である。よってそれは `return` や `jmp` などの特定のアドレスへ移動可能な処理であり、それにプログラムのアドレスを入れることによってその自身の任意のプログラムを実行可能にすることができる。

この対策としては、単純に境界チェックを行う関数を用いる。もしくは C 言語を辞めるという手がある。他のガベージコレクションを持つ言語を利用すればこのような問題は無い。しかし、その分プログラムは遅くなる。しかし、メモリ安全性に関わる問題は非常に多く、大人数を導入したとしても完全な解決になっている訳では無い。事実 Linux 開発にも Rust が順次導入されている [?]. C 言語は完璧な開発者には最適な言語であるが、そのような開発者は現状ではまだ現れていない。やはり、開発者も間違いを犯すという設計思想のもと開発された言語を使用するほうがよりサービス自体に問題が少ないよう感じる。そのため、私は C 言語は言うまでもなく素晴らしい言語であるが、同時に開発者のミスが多大なる被害を巻き起こす。少しづつ他の言語へと移行していくべきであると考える。

参考文献

- [1] https://www.caida.org/catalog/papers/2003_sapphire/
- [2] <https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/>