演習 3

1270360:稗田隼也

2 月 3 日

1 概要

本演習では、バッファオーバーフロー（Buffer Overflow, BOF）の脆弱性について理解を深め、その脆弱性を修正することで攻撃の防止策を学ぶことを目的とした。バッファオーバーフローは、プログラムの不適切なメモリ管理によって発生し、悪意ある攻撃者が任意のコードを実行できる可能性を持つ深刻なセキュリティリスクである。本レポートでは、BOF の基本的な仕組み、実際の攻撃手法、防御策の実装、そして修正後の動作検証について詳述する

2 内容

本演習で使用した脆弱なプログラム hello.c には、ユーザ入力を処理する際に gets() 関数が使用されていた。この関数は入力サイズの制限を行わないため、過剰なデータがバッファに書き込まれた場合に、スタック上のリターンアドレスが上書きされる危険がある。これにより、攻撃者は本来実行されるべきでないコード、例えば shell() 関数を呼び出し、任意のシェルを取得することが可能となる。攻撃の準備として、objdump を用いて shell() 関数のアドレスを特定し、pwntools を利用してペイロードを作成した。ペイロードは、バッファの境界を超えて書き込みを行い、スタック上のリターンアドレスを shell() 関数のアドレスに書き換えることで、意図的にシェルを起動させるものである。また、call 命令と ret 命令の動作を理解することで、スタックの構造と攻撃の仕組みをより深く把握した。この脆弱性を解消するため、gets() を fgets() に置き換え、入力サイズの制限を設けた。さらに、strcpy() を strncpy() に変更し、バッファオーバーフローの発生を防ぐためのコーディングを行った。加えて、プログラムのコンパイル時に以下のセキュリティ対策を施した。スタック保護の有効化（-fstack-protector オプション）実行防止機能（No-Exec Stack）の設定（-z noexecstack オプション）アドレス空間配置のランダム化（ASLR） の適用これらの修正を加えた後、プログラムを再コンパイルし、supervisorctl を用いてサーバプロセスを再起動した。その後、同様の攻撃コード（exploit.py）を実行し、修正前後の挙動を比較した。修正前はシェルを取得することができたが、修正後は入力が制限され、攻撃は失敗に終わったことを確認した。

3 考察

本演習を通じて、バッファオーバーフローの脆弱性がいかに重大なセキュリティリスクを引き起こすかを実体験することができた。特に、入力データの適切な検証が行われない場合、メモリの改ざんによって任意のコードが実行されることがあることを理解した。gets() のような危険な関数の使用は避け、安全な関数（例: fgets(), strncpy()）を使用することで、BOF のリスクを大幅に減少させることが可能である。また、コンパイル時に適切なセキュリティオプションを有効化することで、バッファオーバーフロー攻撃をさらに困難にできることも確認できた。特に、スタック保護、実行防止機能、および アドレス空間配置のランダム化 は、ソフトウェアの安全性を高めるための有効な対策であることがわかった。さらに、本演習では Docker 環境を活用し、実験環境のセットアップや復旧を迅速に行う方法も習得した。これにより、安全な環境下でのセキュリティテストの重要性と実践的な手法を学ぶことができた。

4 まとめ

本演習では、バッファオーバーフローの脆弱性の発見から修正、そして対策の検証までを一貫して行った。gets() の代わりに fgets() を使用し、入力サイズを制限することで、BOF の発生を防ぐことができた。また、脆弱性修正後のプログラムに対して再度攻撃を試みた結果、攻撃は成功しないことが確認された。今回の経験を通じて、バッファオーバーフローの危険性とその対策の重要性を深く理解することができた。今後は、より高度なセキュリティ技術を学び、ソフトウェア開発におけるセキュリティ意識をさらに高め、安全なプログラム設計を実践していきたいと考えている。