バッファオーバーフローの悪用と防御の回避

メモリ破壊の脆弱性、攻撃手法、
防御の回避テクニック、そして防御戦略を
深く掘り下げる

- DEP / NX
- ASLR
- スタックカナリア
- Intel CET
- 制御フロー インテグリティ
- ARM PAC

エクスプロイト開発 | ROP / JOP シェルコード | 脆弱性調査（リサーチ） | ファジング & 解析 | 防御戦略

Steve T.

バッファオーバーフローの悪用と防御の回避

メモリ破壊脆弱性の徹底解説：エクスプロイトと防御

Steve T. Publications

本書はこちらで販売中です
https://leanpub.com/buffer-overflow-exploit-defense-jp

この版は 2026-07-16 に発行されました。

本書は Leanpub の電子書籍です。Leanpub はリーンパブリッシングプロセスで著者や出版社を支援します。リーンパブリッシング は新しい出版スタイルです。軽量なツールを使って執筆中の電子書籍を出版し、読者のフィードバックをもらいながら魅力的な本に仕上がるまでピボットを繰り返すことができます。

Contents

メモリ破壊脆弱性の徹底解説：エクスプロイトと防御

本書について：本書は、メモリ破壊脆弱性について包括的かつ技術的に厳密な扱いを提供します。CPU アーキテクチャの基盤から、高度なエクスプロイト技術、そして今日のセキュリティランドスケープを定義する現代の防御機構までを網羅します。コンピュータのメモリアーキテクチャ、プロセスレイアウト、アセンブリ言語の基礎から始まり、スタックベースおよびヒープベースのバッファオーバーフロー、use-after-free、フォーマット文字列脆弱性、および関連するメモリ破壊バグへと進みます。本書は、シェルコード、リターンエンテッドプログラミング、ジャンプ指向プログラミング、および Windows と Linux の両プラットフォームにおける情報リークなどのエクスプロイト開発コンセプトをカバーします。DEP/NX、ASLR、スタックカニバリ、Intel CET、ARM PAC、コントロールフローインテグリティなどの現代のミティゲーションを詳細に解説し、エクスプロイト研究と防御機構の間の継続的な軍備競争についても取り上げます。最終章では、脆弱性研究メソッド論、ファジング、セキュアコーディングプラクティス、およびインシデントレスポンスについて述べています。本書は、防御目的のためにメモリ破壊を最も深いレベルで理解したいセキュリティプロフェッショナル、リバースエンジニア、脆弱性研究者、そして上級学生を対象としています。記載されているすべての技術は、倫理的セキュリティ研究と責任ある公開の文脈で提示されています。

はじめに

1988 年 11 月、コーネル大学の大学院生ロバート・モリスは、インターネット上に拡散した最初の重大なコンピュータワームとして知られることになったプログラムをリリースしました。それがモリスワームです。このワームは、数千の Unix システムで稼働していた fingerd デーモンにおけるバッファオーバーフロー脆弱性を悪用しました。プログラムの固定サイズバッファが保持できるよりも多くのデータを送信することで、ワームは呼び出しスタック上のリターンアドレスを上書きし、リモートコード実行を達成しました。当時のインターネットに接続されていたコンピュータの推定 10%が感染し、広範囲な混乱を引き起こし、実質的に現代のコンピュータセキュリティ分野を生み出しました。

30 年以上が経過した現在でも、バッファオーバーフロー脆弱性は最も危険なソフトウェアバグのクラスの一つであり続けています。Common Vulnerabilities and Exposures データベースは毎年、境界外書き込み、use-after-free 状態、ヒープオーバーフロー、および関連するメモリ破壊欠陥に関する数千の CVE エントリをリストしています。現代のオペレーティングシステムはこれらの攻撃に対抗するために防御の武庫を配備しています：非実行メモリページ、アドレス空間ランダム化、スタックカニバリ、コントロールフローインテグリティ、そして Intel の Control-Flow Enforcement Technology や ARM の Pointer Authentication Codes のようなハードウェアベースの保護機能です。しかし、攻撃者は各新しいレイヤーを迂回する方法を見続け、エクスプロイト技術とミティゲーションの間の軍備競争は緩む気配を示していません。

本書が存在する理由は、メモリ破壊を根本的なレベルで理解することが、ソフトウェアセキュリティに関わるすべての人にとって本質的に重要だからです。攻撃者がスタックオーバーフロー、ヒープ操作、そしてリターンエンテッドプログラミングペイロードをどのように連鎖させるかを理解できない防御側のプロフェッショナルは、効果的な保護を設計することに苦労するでしょう。脆弱性研究者は、新しいバグを発見し責任ある公開を行うために、メモリレイアウト、アロケータ内部動作、そして呼び出し規約に関する深い知識を必要とします。マルウェアを分析するリバースエンジニアは、その能力を評価するためにエクスプロイトプリミティブを理解しなければなりません。たとえアプリケーション開発者でさえ、特定のコーディングパターンがなぜ危険なのか、そしてコンパイラの強化フラグが実際にはどのように動作するのかを知ることで利益を得ます。

本書は、基礎から高度な技術への段階的な旅として構成されています。初期の章はハードウェアとオペレーティングシステムの文脈を確立します：CPU がメモリとどのように相互作用するか、プロセスが仮想アドレス空間にどのようにレイアウトされるか、アセンブリ言語の命令がレジスタレベルで何を意味するか、そして呼び出し規約がプラットフォーム

間で関数呼び出しをどのように制御するかです。この基盤なしでは、後続のエクスプロイトに関する議論は根拠を欠くことになります。

中間の章は、特定の脆弱性クラスとエクスプロイト技術に深く掘り下げています。スタックベースのバッファオーバーフローは基礎的なメモリ破壊バグとして詳細な扱いを受け、その後、アロケータ内部動作のために著しく複雑なヒープエクスプロイトが続きます。オフバイワンエラー、use-after-free、ダブルフリー、フォーマット文字列欠陥、整数オーバーフロー、および競合状態を含む関連脆弱性タイプが体系的にカバーされます。シェルコード構築、コントロールフロー乗っ取り、リターンエンテッドプログラミング、ジャンプ指向プログラミング、そして sigreturn 指向プログラミングなどのエクスプロイト開発コンセプトはこれらの基盤の上に構築されます。ASLR を無効にする情報リークとアドレス公開技術は、専用の章で解説されています。

後期の章は防御へと移行します。現代のミティゲーションが個別に検証されます：DEP/NX がハードウェアレベルでどのように動作するか、ASLR がどのようにエントロピーを提供しどこで不足するか、スタックカニバリが損害が広がる前に破壊をどのように検出するか、PIE と RELRO がコードセクションをどのように保護するか、そして LLVM、Intel CET、Microsoft CFG、ARM PAC におけるコントロールフローインテグリティ実装がコントロールフロー転送をどのように制限するかです。専用の章は各ミティゲーションとそのバイパス技術の間の歴史的軍備競争をカバーしています。なぜなら、どのような防御が失敗したかを理解することは、それらが何を守っているかを理解することと同じくらい重要だからです。

最終章はより広いエコシステムに取り組んでいます：ファジング戦略とクラッシュトライアージを含む脆弱性研究メソッド論、セキュアコーディングプラクティスとコンパイラ強化オプション、そして本番環境でのアクティブなエクスプロイトに対応する防御者のためのエクスプロイト検出、インシデントレスポンス、およびフォレンジック分析です。

本書全体を通して、コード例は C 言語で提供され、付随するアセンブリリストが脆弱性が命令レベルでどのように現れるかを説明しています。メモリ図は、破壊前後のデータレイアウトを示しています。デバッガ出力は実践的な分析ワークフローを実演しています。Linux と Windows 間のクロスプラットフォーム比較は、共有される原則とプラットフォーム固有のニュアンスの両方を浮き彫りにしています。学術研究が理解を形作った箇所では、発表された論文や会議録への参照が提供されています。

いくつかの重要な免責事項を述べておく必要があります。本書は教育的・防御的目的のためにエクスプロイト技術を記述しています。著者は倫理的セキュリティ研究と脆弱性の責任ある公開を強く支持しています。読者が所有していないソフトウェアでメモリ破壊バグを発見した場合、確立された責任ある公開プロセスに従うべきです：ベンダーに非公開で通知し、修正が開発されるために合理的な時間を認め、脆弱性がパッチ適用された後にのみ詳細を公開してください。明示的な許可なしにシステムに対してエクスプロイト技術を使用することは、ほとんどの管轄区域で違法であり、いずれにせよ倫理的ではあ

りません。

本書は、ポインタ、配列、構造体、そして malloc および free を用いた動的メモリ割り当てを含む基本的な C プログラミングへの精通を前提としています。事前のアセンブリ言語の経験やエクスプロイト開発のバックグラウンドは必要ありません。これらのトピックはゼロから構築されます。焦点は x86_64 アーキテクチャに置かれており、これは主要なデスクトップおよびサーバープラットフォームです。ARM64 はモバイルおよび組込みコンテキストに関連する箇所で議論されています。Linux と Windows が例の主要なオペレーティングシステムとして機能していますが、多くの概念は広範に適用されます。

本書の終わりまでに、読者はメモリ破壊を孤立したトリックの集合体としてではなく、コンピュータアーキテクチャ、オペレーティングシステム設計、そしてソフトウェアエンジニアリングに根ざした一貫した分野として理解できるようになります。その理解こそが、効果的な攻撃的研究と堅牢な防御的エンジニアリングの両方が構築される基盤です。

第 1 章：コンピュータメモリアーキテクチャと CPU 基礎

メモリ破壊を理解するには、それを可能にするハードウェアを理解する必要があります。すべてのバッファオーバーフロー、use-after-free、フォーマット文字列エクスプロイトは、最終的にプログラムがメモリレイアウトについて仮定することと、CPU が実際にデータをどのように処理するとの間の不一致を悪用しています。本章はアーキテクチャの基盤を確立します。

ヨン・ノイマンアーキテクチャとメモリアドレッシング

現代のコンピュータは、1945 年にジョン・フォン・ノイマンによって提唱されたフォン・ノイマンアーキテクチャに従っています。このモデルでは、命令とデータは同じメモリ空間を共有し、同じバスを介してアクセスされます [9]。CPU はメモリから命令を取り込み、デコードし、実行し、結果をメモリに保存します。この統合されたメモリモデルは、コンピューティングの柔軟性の源でありながら、根本的なセキュリティ脆弱性の源でもあります：データが CPU が命令を見つけることを期待する場所に配置できる場合、攻撃者はデータを破壊することで任意のコードを実行できます。

メモリは、アドレス可能バイトの線形配列として組織されています。各バイトはゼロから始まる一意の数値アドレスを持っています。32 ビットシステムでは、アドレス範囲は 0x00000000 から 0xFFFFFFFF（4 ギガバイト）です。64 ビットシステムでは、理論的な範囲は 2^64 バイト（16 エクサバイト）に拡張されますが、実際の実装はより少ないアドレスビットを使用しています。x86_64 は現在 48 ビットの仮想アドレスを使用しており、プロセスあたり 256 テラバイトのアドレス空間を提供しています。

CPU は、すべてのメモリ操作時に連携して動作する 3 つの相互接続されたバスを介してメモリにアクセスします。アドレスバスは、CPU が読み取りまたは書き込みを行いたいメモリアドレスを負担し、プロセッサからメモリサブシステムへ一方向に流れます。x86_64 では、アドレスバスの幅が CPU が到達できる一意の物理アドレスの数を決定します。データバスは実際に転送されるバイトを負担し、双方向に動作します。読み取り操作時にはデータがメモリから CPU へ流れ、書き込み時には逆方向に流れます。コントロールバスは、現在の操作が読み取りか書き込みか、メモリサブシステムがデータの受け入れ準備ができているか、そして転送が完了したタイミングを示すタイミングおよび制御信号を負担することで転送を調整します。

バッファオーバーフローエクスプロイト中、これらのバスは攻撃全体を通してアクティブです。オーバーフローするstrcpy がバッファの境界を超えて書き込むと、各バイトがデータバスを介してスタック上のターゲットアドレスへと移動します。コピーがバッファから保存されたフレームポインタおよびリターンアドレスへと進捗するにつれて、アドレスバスは次第に高いスタックアドレスを提供します。オーバーフローデータを制御する攻撃者は、実質的にこれらのバスを通過する値を制御し、その延長線上で、CPU が RET 命令を実行した際に後で読み取る値を制御します。このハードウェアレベルのデータフローを理解することで、メモリ破壊がどのようにして直接コントロールフロー乗っ取りに翻訳されるかが明確になります：CPU には正当なスタックデータとたまたま有効なアドレスのように見える攻撃者供給バイトとの区別がつきません。

x86_64 におけるメモリアドレッシングは複数のモードをサポートしています。即値アドレッシングは命令に直接定数値を埋め込みます。レジスタアドレッシングはレジスタの内容をオペランドとして使用します。直接メモリアドレッシングは絶対アドレスを指定します。最も一般的には、ベースプラス変位アドレッシングはベースレジスタの値に変位を加算して有効アドレスを計算し、オプションでインデックスレジスタと 1、2、4、8 の係数でスケーリングされます。この最後のモードは配列へのアクセス方法です：ベースレジスタが配列の開始アドレスを持ち、インデックスレジスタが要素番号を持ち、スケールファクタが要素サイズを考慮します。

CPU レジスタとその役割

レジスタは CPU 内部の小型で高速な記憶領域です。プロセッサダイ上に存在するため、メインメモリよりも桁違いに高速です。アセンブリ言語のすべての命令はレジスタまたはレジスタを介してアドレス指定されたメモリに対して動作します。エクスプロイト開発において、各レジスタが何をするかそして呼び出し規約によってどのように使用されるかを理解することは重要です。

x86_64 アーキテクチャは 16 個の汎用 64 ビットレジスタを提供しています [13, 91]：

RAX（累算レジスタ）：伝統的に算術演算および関数の戻り値に使用されます。syscall インタフェースでは、RAX がシステムコール番号を保持します。名称は AX（16 ビット）に由来し、AX は AL と AH（8 ビットの下位半分と上位半分）に由来します。32 ビットの EAX への書き込みは RAX の上位 32 ビットをゼロ化します。この性質はシェルコードで null バイトなしペイロードを生成するために悪用されます。

RBX（ベースレジスタ）：しばしばデータアクセスのベースポインタとして使用されます。System V AMD64 ABI では、RBX は callee-saved レジスタです。これを修正する関数はリターン前に元の値を復元する必要があります。

RCX（カウンタレジスタ）：歴史的に文字列および反復命令におけるループカウンタとして使用されました。Windows x64 呼び出し規約では、RCX が最初の関数引数を保持します。

RDX（データレジスタ）：I/O 操作および乗算・除算結果のセカンダリ累算レジスタ（RDX:RAX に保存される 128 ビット結果を生成）として使用されます。

RSI（ソースインデックス）と RDI（デスティネーションインデックス）：文字列演算でソースおよびデスティネーションアドレスを指定するために使用されます。System V AMD64 ABI では、RSI と RDI がそれぞれ第一および第二の関数引数を保持します。

RSP（スタックポインタ）：現在のスタックフレームのトップを指します。x86 ではスタックは下向きに成長するため、値をプッシュすると RSP が減算されます。このレジスタはエクスプロイトの中心にあります。なぜなら RSP を制御することは、CPU がリターン命令を実行する際に読み取るデータを制御することになるからです。

RBP（ベースポインタ／フレームポインタ）：慣習的にスタックフレーム内の安定した参照点として使用されます。コンパイラは`-fomit-frame-pointer` フラグで最適化して削除できますが、ほとんどのデバッグビルドおよび多くのプロダクションビルドはまだ RBP を使用してスタックフレームの連結チェーンを作成し、バックトレース機能を有効にしています。

R8 から R15：64 ビット拡張で追加された 8 つの追加汎用レジスタです。System V ABI では、R8 と R9 が第五および第六の関数引数を保持します。Windows x64 では、R8 と R9 が第三および第四の引数を保持します。

汎用レジスタに加え、いくつかの特殊目的レジスタが関連します：

RIP（インストラクションポインタ）：次に実行する命令のアドレスを保持します。これはコントロールフロー乗っ取り攻撃の主要ターゲットです。他のレジスタとは異なり、RIP は直接書き込むことができません。call、jump、return、および interrupt 命令によってのみ修正されます。RIP を制御することはプログラム実行を制御することを意味します。

RFLAGS（フラグレジスタ）：算術および論理演算によって設定される条件コードを含みます：ゼロ結果のための Zero Flag (ZF)、レジスタ幅を超えたオーバーフローのための Carry Flag (CF)、負の結果のための Sign Flag (SF)、符号付オーバーフローのための Overflow Flag (OF)、そして文字列演算の方向を制御する Direction Flag (DF) です。条件ジャンプ命令はこれらのフラグをテストして分岐するかどうかを決定します。

命令実行とフェッチ・デコード・実行サイクル

CPU は反復するサイクルで命令を実行します：

フェッチ：CPU は RIP が保持するアドレスのメモリから次の命令を読み取ります。x86_64 では、命令は可変長（1〜15 バイト）です。そのため、CPU は次の命令に進める前に最初のバイトをデコードして命令長を決定する必要があります。

デコード：フェッチされたバイトは、実行ユニットが理解する内部マイクロオペレーションに変換されます。現代のアウトオブオーダープロセッサはサイクルごとに複数の命令をデコードし、並列実行のためにそれらを再配置することがあります。

実行：デコードされた演算が実行されます。これにはレジスタまたはメモリからオペランドを読み取り、ALU で算術を実行し、値を比較し、フラグを更新する作業が含まれる可能性があります。

ライトバック：結果がデスティネーションレジスタまたはメモリに書き戻されます。

コミット：アウトオブオーダープロセッサでは、結果はプログラム順序でアーキテクチャ状態にコミットされ、命令がアウトオブオーダーで実行されたとしても観測可能な動作が逐次実行と一致することを保証します。

エクスプロイトにとっての重要な洞察は、CPU が RIP が指すアドレスを信頼するという事実です。ハードウェアレベルで「正当な」コードと「悪意のある」コードの本質的な区別はありません。バッファオーバーフローがスタック上のリターンアドレスを上書きし、上書きされた値が有効な命令バイトを含む攻撃者制御データを指す場合、CPU は疑うことなくそのバイトを実行します。これはすべてのコード実行エクスプロイトの根本前提です。

いくつかの x86_64 命令がエクスプロイト開発に特に関連しています：

- **MOV**：レジスタ、メモリ、即値の間でデータをコピーします。
- **PUSH / POP**：値をスタックにプッシュ（RSP 減算）またはスタックからポップ（RSP 加算）します。
- **CALL**：現在の RIP をスタックにプッシュし、ターゲットアドレスへジャンプします。関数呼び出しに使用されます。
- **RET**：スタックのトップを RIP にポップします。これがリターンエンテッドプログラミングガジェットを連鎖させる命令です。
- **JMP**：ターゲットアドレスへの無条件ジャンプ。ジャンプ指向プログラミングで使用されます。
- **XOR / AND / OR**：ビット単位の論理演算。シェルコードで null バイトなしの値初期化に頻繁に使用されます。
- **LEA（有効アドレス読み込み）**：実際にメモリにアクセスせずにアドレスを計算します。算術および位置独立コードで現在の RIP を取得するのに有用です。
- **SYSCALL / INT 0x80**：システムコールを呼び出すためにユーザーモードからカーネルモードへ移行します。

エンディアンとデータ表現

エンディアンは、マルチバイト値がメモリに保存される際のバイト順序を決定します。x86_64 はリトルエンダンの順序を使用します：最下位バイトが最も低いアドレスに格納されます。例えば、64 ビット値 0x4142434445464748 は以下のように保存されます：

アドレス	バイト
0x1000	0x48 (‘H’)
0x1001	0x47 (‘G’)
0x1002	0x46 (‘F’)
0x1003	0x45 (‘E’)
0x1004	0x44 (‘D’)
0x1005	0x43 (‘C’)
0x1006	0x42 (‘B’)
0x1007	0x41 (‘A’)

バッファオーバーフローをエクスプロイトする際、エンディアンは重要です。なぜなら攻撃者はマルチバイト値（リターンアドレスなど）を正しいバイト順序で供給しなければならないからです。リトルエンダンのシステムでは、印刷された見た目と比較して逆順のアドレスバイトが必要です。

これは部分的な上書きにも影響します。攻撃者が 64 ビットポインタの最下位バイトのみを上書きできる場合、リトルエンダンのシステムではそのバイトは最も低いアドレスにあり、値の下位 8 ビットを表します。ビッグエンダンのシステムでは、同じバイトが上位 8 ビットを表すことになります。

特権レベルとリングアーキテクチャ

x86 プロセッサは 4 つの特権レベルを実装しています。ring 0 から 3 までで、ring 0 が最も特権が高く、ring 3 が最も低いです。現代のオペレーティングシステムは通常 2 つしか使用しません：

- **Ring 0（カーネルモード）**: オペレーティングシステムのカーネルがここで実行されます。すべてのハードウェア、メモリ、CPU 機能に無制限アクセスがあります。カーネルモードコードは任意の物理メモリアドレスを読み書きし、ページテーブルエントリを修正し、CLI（割り込みフラグクリア）や LGDT（グローバルディスクリプタテーブル読み込み）のような特権命令を実行できます。

- **Ring 3（ユーザーモード）**：アプリケーションプログラムがここで実行されます。ユーザーモードコードはハードウェアまたはカーネルメモリに直接アクセスできません。特権命令の実行またはカーネルアドレスへのアクセスの試みは一般プロテクトフォールトを引き起こし、カーネルはそのプロセスを終了することでこれを処理します。

リング間の移行はハードウェアによって制御されます。システムコールは SYSCALL 命令（x86_64）または INT 0x80（レガシィ x86）を使用して ring 3 から ring 0 へ切り替えます。割り込みと例外もリング移行を引き起こします。CPU はカーネルハンドラに制御を移す前に、RIP および RFLAGS を含む現在の状態をカーネルスタックに保存します。

この特権分離はエクスプロイトに関連します。なぜならユーザーモードのエクスプロイトは ring 3 でできることに制限されるからです。ユーザーアプリケーションにおける成功したバッファオーバーフローはそのプロセスのコントロールフローを乗っ取れますが、他のプロセスのメモリやカーネルデータに直接アクセスすることはできません。特権を昇格させるために、攻撃者はカーネル自体の脆弱性（カーネルモードのバッファオーバーフロー）を悪用するか、システムコールインタフェースを乱用する方法を見つける必要があります。

ハードウェアレベルでのメモリマッピング

Memory Management Unit (MMU) は、ソフトウェアが使用する仮想アドレスを RAM が使用する物理アドレスに変換するハードウェアコンポーネントです [11]。MMU はオペレーティングシステムによって管理される階層型データ構造であるページテーブルを使用してこの変換を実行します [10, 14]。

x86_64 の 4 レベルページングでは、48 ビット仮想アドレスは以下のように分割されます：

- ビット 47:39（9 ビット）：Page Map Level 4 (PML4) テーブルのインデックス
- ビット 38:30（9 ビット）：Page Directory Pointer Table (PDPT) のインデックス
- ビット 29:21（9 ビット）：Page Directory (PD) のインデックス
- ビット 20:12（9 ビット）：Page Table (PT) のインデックス
- ビット 11:0（12 ビット）：4-KiB ページ内のオフセット

各ページテーブルレベルは 512 エントリ（2^9）を含み、各エントリは 8 バイトです。CPU は CR3 レジスタに保存されたルートポインタから始めてこれらのテーブルをウォークし、仮想アドレスに対応する物理フレームを見つけます。

各 Page Table Entry (PTE) には物理フレーム番号だけでなく、権限フラグも含まれます：

- **Present (P)**：ページが現在物理メモリにあるかどうか

- **Read/Write (R/W)**：書き込みが許可されているかどうか
- **User/Supervisor (U/S)**：ユーザーモードコードがページにアクセスできるかどうか
- **Execute Disable (NX/XD)**：このページからのコード実行が許可されているかどうか

AMD の拡張ページテーブルと Intel の Execute Disable 機能で x86_64 に追加された NX ビットは、DEP のハードウェア基盤です [5, 6]。ページテーブルエントリに NX ビットが設定されている場合、そのページからの命令実行の試みはすべてプロテクトフォールトを引き起こします。これによりスタックまたはヒープに配置されたシェルコードの直接実行を防ぎます。

Translation Lookaside Buffer (TLB) は、最新の仮想から物理への変換をキャッシュし、すべてのメモリアクセスでマルチレベルページテーブルウォークを行うことを回避します。TLB は通常小型（現代プロセッサでは 64〜128 エントリ）なので、プロセスのマッピングのサブセットのみを保持します。TLB ミス時には、CPU はハードウェアページウォークを実行し、数十サイクルがかかります。

ページレベル権限を理解することはエクスプロイトに本質的です。なぜなら多くの技術が特定の権限組み合わせを持つメモリ領域を見つけることに依存しているからです。リターンエンテッドプログラミングは実行可能コードページ（テキストセグメント）がすでに実行可能としてマークされているからこそ動作します。攻撃者は新しくコードを書込可能だが非実行の領域に注入するのではなく、既存のコードを再利用します。

第 2 章：プロセスメモリレイアウトとオペレーティングシステムメモリ管理

仮想メモリとアドレス変換

テキスト、データ、BSS、ヒープ、スタックセグメント

共有ライブラリと動的リンク

メモリ割り当て：malloc、free、アロケータ

ページテーブルと MMU

クロスプラットフォームの違い：Linux 対 Windows メモリレイアウト

第 3 章：アセンブリ言語の基礎と呼び出し規約

x86 および x86_64 命令セットの概要

エクスプロイト開発における一般的な命令

スタックフレーム：プロローグ、エピローグ、およびローカル変数

System V AMD64 ABI 呼び出し規約

Windows x64 呼び出し規約

位置独立コードとリロケーション

第 4 章：デバッグワークフローとバイナリ分析の基礎

セキュリティ研究における GDB の基礎

Windows 上の WinDbg、x64dbg、および Mona.py

メモリ検査とブレークポイント戦略

radare2 および Ghidra によるリバースエンジニアリング

静的解析：objdump, readelf, および PE ツール

Frida による動的解析と計装

第 5 章：スタックベースのバッファオーバーフロー

スタックベースオーバーフローの解剖

リターンアドレス上書きとコントロールフロー乗っ取り

オフセット検索：パターン作成とクラッシュ分析

NOP スレッドとシェルコード配置

環境変数とスタックスプレーイング

Windows および Linux システムへの影響

ワークスルー：シェルコード注入による完全なスタックオーバーフローエクスプロイト

脆弱なプログラム

ステップ 1：保護を無効にしてコンパイル

ステップ 2：GDB およびサイクリックパターンでオフセットを検索

ステップ 3：シェルコード用のスタックアドレスを検索

ステップ 4：シェルコードを用意

ステップ 5：Python エクスプロイトスクリプト

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/buffer-overflow-exploit-defense-jp.

ステップ 6：GDB で検証

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/buffer-overflow-exploit-defense-jp.

主要な観察事項

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/buffer-overflow-exploit-defense-jp.

第 6 章：ヒープベースのバッファオーバーフローとヒープ操作

メモリアロケータの動作：glibc ptmalloc および Windows ヒープ

ヒープチャンクメタデータとフリーリスト管理

ヒープオーバーフローエクスプロイト技術

ビンソートと統合攻撃

House of Force, House of Spirit, および命名ヒープ技術

ワークスルー：任意書きプリミティブによる Fastbin 攻撃

脆弱なプログラム

エクスプロイト戦略：__free_hook 上書きへの Fastbin 攻撃

GDB で必要なアドレスを検索

Python エクスプロイトスクリプト（pwntools）

GDB でエクスプロイトを検証

第 7 章：関連するメモリ破壊脆弱性

オフバイワンエラーとそのエクスプロイト

メモリ破壊につながる整数オーバーフロー

Use-After-Free 脆弱性

ダブルフリーと解放ポインタ操作

フォーマット文字列脆弱性

境界外読み取りと書き込み

メモリセーフティに影響する競合状態

ワークスルー：**vtable** 上書きによる **Tcache** ポイズニング **UAF**

脆弱なプログラム

エクスプロイト戦略：**vtable** 上書きへの **Tcache** ポイズニング

GDB で必要なアドレスを検索

Python エクスプロイトスクリプト（**pwntools**）

GDB で検証

ワークスルー：GOT 上書きによるフォーマット文字列任意書き込み

脆弱なプログラム

エクスプロイト戦略：libc をリークし GOT を上書き

必要なアドレスを検索

Python エクスプロイトスクリプト（pwntools）

GDB で検証

第 8 章：シェルコードの基礎とコントロールフロー乗っ取り

シェルコードとは何か、そしてなぜそれが重要か

位置独立シェルコードの書き方

Linux 用システムコールベースシェルコード

Windows API 解決と LoadLibrary 技術

Null バイト処理とエンコーディング

ポリモルフィックおよびエンコードペイロード

ワークスルー：スタックオーバーフローでのシェルコード構築および実行

脆弱なプログラム

シェルコード: execve(“/bin/sh”)

GDB でオフセットを検索

Python エクスプロイトスクリプト

GDB で検証

第 9 章：リターンエンテッドプログラミングとコード再利用攻撃

DEP が解決した問題と ROP ソリューション

ROP チェーンの構築：ガジェット、ピボット、レジスタ

ropper および ROPgadget によるガジェット検索

Linux 上の ROP：システムコール構築

Windows 上の ROP：VirtualAlloc および WinExec

ワークスルー：強化されたバイナリに対する完全な ROP チェーン

脆弱なプログラム（強化済み）

ステップ 1：保護を検証しガジェットを見つける

ステップ 2：libc アドレスを見つける

ステップ 3：ROP チェーンを構築

ステップ 4：Python エクスプロイトスクリプト

ステップ 5：GDB で検証

この実演からの主要な教訓

ジャンプ指向プログラミング（JOP）

sigreturn 指向プログラミング（SROP）

Oriented Return Composition（ORC）

第 10 章：情報リークとアドレス公開

なぜ ASLR がエクスプロイトを困難にするか

ヒューブスプレーイングと決定論的メモリアウト

フォーマット文字列情報開示

return-to-PLT および GOT 上書きによるリーク

ワークスルー：ASLR バイパスによるマルチステージエクスプロイトチェーン

脆弱なプログラム（**ASLR + Partial RELRO**）

2 段階攻撃の理解

Python エクスプロイトスクリプト

オフセット数学の理解

Windows ASLR バイパス技術

部分的な上書きと減少エントロピー

クロスプロセス情報収集

第 11 章：現代の防御とミティゲーション

DEP/NX: 非実行メモリページ

ASLR: アドレス空間レイアウトランダム化

スタックカニバリおよび ProPolice

PIE, RELRO, および GOT 保護

SafeSEH および CFG（Windows）

コントロールフローインテグリティ（CFI）

Intel CET: シャドウスタックおよび間接ブランチトラッキング

ARM ポインタ認証（PAC）

サンドボックス化およびプロセス分離

第 12 章：バイパス研究とエクスプロイト-ミティゲーション軍備競争

タイムライン：スタックスマッシングから現代の防御へ

カニバリバイパス技術

ASLR エントロピー分析およびバイパス

Full RELRO および GOT 書き込み制限

CFI バイパス研究

CET バイパスアプローチ

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/buffer-overflow-exploit-defense-jp.

ARM 上の PAC バイパス

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/buffer-overflow-exploit-defense-jp.

未来：何が次に来るか

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/buffer-overflow-exploit-defense-jp.

第 13 章：脆弱性研究メソッド論

ファジング戦略：カバレッジガイド型およびミューテーションベース

ワークスルー：ファジングキャンペーンのセットアップおよび実行

ターゲットプログラム：脆弱な JSON 風パーサー

オプション A: AFL++ でのファジング

オプション B: libFuzzer でのファジング

カバレッジの解釈およびキャンペーンの最適化

クラッシュトライアージと根本原因分析

バイナリ分析：象徴的実行およびコンコリックテスト

ワークスルー：**Angr** でパスワードチェックを解決

メモリサニタイザ: **ASan, MSan, UBSan**

差分ファジングおよびクロスプラットフォーム比較

責任ある公開および **CVE** 割り当て

第 14 章：セキュアコーディングプラクティスとコンパイラ強化

安全な文字列およびメモリ関数

境界チェックおよび安全な整数算術

コンパイラフラグ: FORTIFY_SOURCE, StackProtector, CFProtection

メモリセーフ言語: Rust を代替として

メモリセーフティのためのコードレビュー技術

静的解析ツール: Clang Analyzer, Coverity, PVS-Studio

第 15 章：エクスプロイト検出、インシデントレスポンス、およびフォレンジック

野良でのバッファオーバーフロー攻撃の検出

EDR およびメモリ破壊のための振る舞いモニタリング

コアダンプ分析および死後デバッグ

ネットワークレベルエクスプロイト検出

フォレンジックタイムライン再構築

主要エクスプロイトインシデントからの教訓

ワークスルー：スタックオーバーフローエクスプロイトのフォレンジック分析

シナリオ: 侵害されたウェブサービス

ステップ 1: GDB によるコアダンプ分析

ステップ 2: エクスプロイトアーティファクトのためにスタックを検査

ステップ 3: 脆弱な関数を特定

ステップ 4: Volatility によるメモリイメージ分析

ステップ 5: メモリからネットワーク接続を抽出

ステップ 6: 攻撃タイムラインを再構築

ステップ 7: 帰属および侵害インジケータ

このフォレンジック分析からの主要教訓

結論

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/buffer-overflow-exploit-defense-jp.

参考文献