2007－DPS－133 (10)
2007／11／27

リモートバッファオーバーフロー攻撃の検出に関する研究

李 碧波† 上原 稔† 森 秀樹†
†東洋大学大学院工学研究科情報システム専攻

リモートバッファオーバーフローは、ネットワークセキュリティ領域で最も深刻なセキュリティホールである。リモートバッファオーバーフロー攻撃に防御能力を強化することは、重要な研究課題になっている。本研究は、ハードウェアによるリモートバッファオーバーフロー攻撃の検出手法を提案した。提案手法について、設計方針、基本動作及び各モジュールの基本構造を提示した。最後に、提案手法の性能に影響を与える主な原因について、分析を行った。

An Approach to the Prevention of Remote Buffer Overflow Attacks

Li Bibo† Minoru Uehara† Hideki Mori†
†Department of Open Information Systems Graduate School of Engineering Toyo University

Abstract The remote buffer overflow vulnerability is the most serious security vulnerability in network security domain. How to enhance the defense ability to remote buffer overflow attack becomes an important research subject. In this paper, we put forward a model of defense remote buffer overflow attack based on hardware. The constructing idea and the structure of the model are given, and the realization technology and method of each module are described. In the last of this paper, we give an analysis and remark to the influence of model performance caused by each factor.

1．はじめに

近年、情報ネットワーク技術の発展と普及に伴い、電子商取引、オンラインバンキング、電子政府などの重要な情報やサービスがネットワークにつながるようになってきた．一方、アプリケーションの多様化・複雑化、管理が不十分なコンピュータの増加、不正アクセスにより、情報セキュリティのリスクが増大している。またウイルス、ワーム、トロイの木馬などの不正プログラムによるシステムの停止や情報漏えいなどの被害が深刻化しつつある。このような状況の中、情報セキュリティ対策を強化することの重要性が高まっている。

CERT[1]の統計によれば、バッファオーバーフローに起因するセキュリティホールは、全体の 50%－60%に占めており、比較的容易に悪用される。標準的なスタックオーバーフローを利用すれば、非常に強力なインターネットワームを作成することができる（例：Blaster、Slammer、CodeRed、Nimda など）。また、バッファオーバーフローによって実行される悪性コードは管理者レベルの権限で実行されることが多いため、サーバに対して任意の操作が可能になる。ゆえに、バッファオーバーフロー攻撃を防ぐことは、多くの侵入行為を防ぐことに繋がるため非常に重要である。

バッファオーバーフロー攻撃の動的検出手法として、スタック保護やソフトウェアによるコードスキャンなどが提案されている。理論上ではどれも攻撃に一定な防御効果はあるが、実用性や汎用性に関してはまだ欠陥が残る。

本研究では、ハードウェアによるリモートバッファオーバーフロー攻撃の検出手法を提案する。SimpleScalar [2]ツールセットver3.0dを用いて、本手法を実装し、攻撃パケットDARPA 1999 Training Data[3]を用いて性能への影響を検証した。

2．バッファオーバーフロー攻撃

2.1 バッファオーバーフロー攻撃の原理

バッファオーバーフローとはプログラムの変数用に用意された領域に、領域の大きさを越えたデータが入力されることで発生する脆弱性のことで、最も危険なスタックオーバーフローが発生すると、リターンアドレスが書き換えられ、悪性コードが実行可能になる。その結果、システム全体がのっとられてしまう可能性が発生する。

悪性コード（ShellCode[4]ともいう）は脆弱性を利用する際に実行するコードのことで、攻撃者によってターゲットプログラムに注入され、不正操作を行う。ShellCode の主な用途はシステムコールを実行し、shell を返すことである（例：exec( )を利用し、“/bin/sh”を実行する）。Shell はオーバーフローさせたプログラムと同じ権限であるため、root 権限で suid のプログラムが脆弱性を持つ場合、攻撃者は root 権限の shell を獲得することになる。バッファオーバーフローは攻撃者にとっては、リモート攻撃を行うすべて条件が揃っている：①プログラムのアドレス空間に実行可能な悪性コード（ShellCode）を注入、②脆弱性を利用してバッファをオーバーフローさせ、リターンアドレスを書き換え、プログラム流れを変える、③悪性コードが実行され、shell 獲得。

2.2 ShellCode の構造分析

通常の ShellCode は構造によって、3 種類（NSR 型、RNS 型、AR 型[4]）に分かられる（図１）。種類によって、適用するバッファ環境が異なる。AR 型は環境変数型ともいわれ、事前に ShellCode を環境変数に置く必要があるため、リモート攻撃には使えない型である。

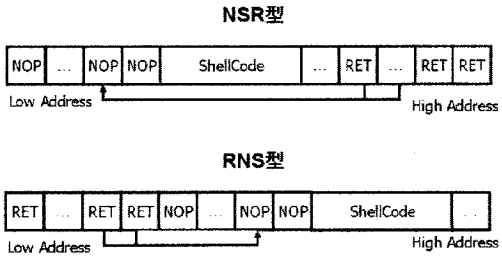

図 1. ShellCode の構造
（N: NOP、S: ShellCode、R: リターンアドレス）

2.3 バッファオーバーフロー攻撃の分類

バッファオーバーフロー攻撃は攻撃源によって、リモートとローカルの 2 種類に分けられる。

リモートバッファオーバーフローは、外部に向かって公開されているサービス（いわゆるサーバ）のバッファオーバーフロー脆弱性を利用して、そのサーバのユーザ権限を奪うために行われる。ネットワーク及びホストコンピュータのセキュリティ措置（ファイアウォール、IDS、アンチウイルスゲートウェイなど）は、未公開のリモートバッファオーバーフロー攻撃を防ぐには、非常に困難である。

リモートバッファオーバーフローにより一般ユーザとして外部から侵入し、ローカルバッファオーバーフローを利用して root 権限を奪

うというように段階を踏んで行われることもある。

3．関連研究

リモートバッファオーバーフロー攻撃を防ぐ代表的な手法として、大きく3つに分けられる。

3.1 スタック保護による検出技術

StackGuard [5]やStackShield[6] はgcc のパッチという形で提供されている。Stackguard はプログラムコンパイル時にスタックオーバーフローを防ぐコードを追加する。StackShield はプログラムコンパイル時に、リターンアドレスをデータ領域にコピーする。これらの手法は、コンパイル済みのプログラムには適用できず、ソースコードが必ず必要になる。

3.2 バイナリコードの静的分析

バイナリ分析は公開済みの悪性コードの検出には、簡単かつ高速などの利点がある。新たな脆弱性攻撃が現すと、即座に攻撃サンプルを分析し、取り出した特徴コードをライブラリに追加する。これらの手法は、未公開のバッファオーバーフロー攻撃には検出できず、また、既存 ShellCode の変形（Polymorphic ShellCode[4]）が使用される場合、検出が困難になる。

3.3 IDS (Intrusion Detection System) による検出手法

IDS は侵入検知システムのことで、ホスト型IDS（HIDS）は、監視対象サーバに直接インストールして使用する。ホストに流れてきたパケットやユーザの操作情報などを監視することができる。一方のネットワーク型 IDS（NIDS）とは、ネットワーク上に流れるパケットを収集し、不正パケットを検知する。トラフィック量が多い場合は、パケットを収集する処理能力が追いつかず、不正パケットを取りこぼす問題がある。IDS による手法はいずれも、検出ルールを常にバージョンアップしないと、ShellCode の変形には対処できない。

4．提案手法

リモートバッファオーバーフロー攻撃に至る基本条件として、サーバプログラムが外部からのユーザデータを受け取る際、データ処理エラーが発生し、サーバプログラムのプロセスが乗っ取られる。攻撃の最終結果は、悪性コード（ShellCode）が含まれているユーザデータを、サーバプロセスの命令列と見なし、実行されることで、ホストの制御権限が獲得される。したがって、リモートバッファオーバーフロー攻撃を防ぐ最も有効手段は、ユーザデータに含まれている悪性コードの実行を阻止することである。本研究の提案手法は、CPU の L1 命令キャッシュに入る命令列を監視することによって、正確かつ迅速に悪性コードの実行を防ぎ、結果的にリモートバッファオーバーフローのような潜在的な攻撃を阻止することに繋がる。

具体的に、ホストプログラムがユーザから受け取ったデータを、一定サイズに分割して HASH 値を計算する。計算された値を専用バッファへ保存する。同時に、CPU 命令キャッシュを監視し、実行待ちの命令列の HASH 値を計算する。両方向の HASH 値を比較することによって、悪性なリモート情報を命令列として実行しようとする動作が検出され、攻撃を防ぐことができる。

4.1 基本構造

CPU アーキテクチャに２つの処理ユニット（HashKey 計算ユニット、L1 命令キャッシュ監視ユニット）を追加する

①HashKey 計算ユニット：

このユニットに一定サイズのメモリ空間（HASH Buffer）が割り当てられ、FIFO 方式で、HashKey が保存される。Hash 計算ユニットは、ホストサーバプログラムが受け取ったデータを指定サイズに分割し、計算された HashKey を Hash Buffer に保存する。通常のバッファオーバーフロー攻撃は、オーバーフローさせた直後に悪性コードが実行されることから、メモリ空間の利用とコード検出を効率よくするために、Hash Buffer に保存された HashKey に生存時間を設ける。一定の時間を超えると、HashKey がキューから取り出される。

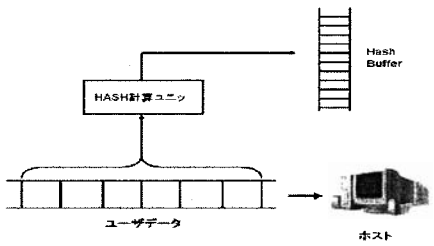

図 2. HashKey 計算ユニット

② L1 命令キャッシュ監視ユニット[7]：

Hash 制御ユニットは CPU の PipeLine 制御パスに接続し、命令キャッシュ中の命令列の commit と更新を監視する。命令列が命令キャッシュに commit 完了すると、Hash 計算ユニットは命令キャッシュ中の実行待ち命令列を読み取り、Hash 値を計算し、Hash Buffer 中の HashKey と比較を行う。一致する場合は、ホストサーバプログラムがリモートデータを受け取り、命令キャッシュに入れ、実行を待つ状態である。これはホストの安全を害する危険行為と見なされ、Hash 制御ユニットは CPU の実行停止割り込み命令を呼び出す。命令キャッシュ中の命令列の実行を阻止すると同時に、CPU の制御パスを通じて、プログラムの実行コンテキストと CPU の割り込み状態を L1 D-Cache に保存する。

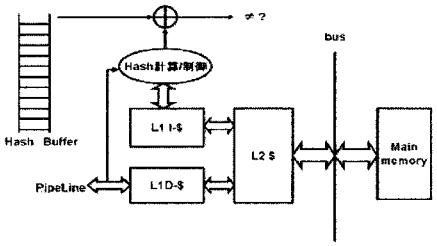

図 3. L1 I-Cache 監視ユニット

4.2 HashKey の計算方法

コードの完全性を確かめるため、暗号を用いてのデータ検証を行う必要がある。任意の長さの入力メッセージを受け取り、高速に動くことから、MD5 アルゴリズムを採用した。HashKey 計算ユニットは受け取ったアプリケーションパケットを 128 ビットで分割し、不足のビットは 0 で補足する。分割されたデータブロックを MD5 アルゴリズムで HashKey を取る。暗号化された HashKey を FIFO 順に Hash Buffer に保存する。L1 I-Cache 監視ユニットも同じ MD5 で I-Cache の命令列を暗号化する。

5．提案手法の実装結果と性能分析

本研究で提案する手法がリモートバッファオーバーフロー攻撃に対して、有効であるかどうかを調べるために、SimpleScalar ツールセ

ットを用いて提案手法を実装し、攻撃パケット DARPA 1999 Training Data を用いて性能への影響を検証した。

表１．実験環境パラメータ

Parameter	Value
L1 I-Cache	16kB, 4-way, 32B cache lines
L1 D-Cache	16kB, 4-way, 32B cache lines
L2 Cache	1MB, 4-way, 32B cache lines
L1 latency	1 cycles
L2 latency	10 cycles
Bandwidth	100 Mbps
Packet	10 Mbps

5.1 分割サイズと検出成功率

HashKey の生存時間を３秒に設定する。データの分割サイズが検出成功率への影響は図４に示されている。

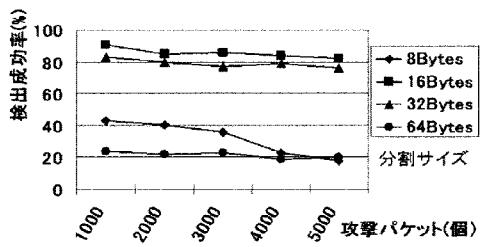

図 4. 分割サイズが検出成功率への影響

実験データから以下の結論が得られる：HashKey の生存時間が固定な場合、データの分割サイズが検出成功率に大きな影響を与える。ShellCode の構造と特徴から、通常な攻撃コードはサイズが短く、かつ多数の組合せ方式がある。データの分割サイズを大きく（64bytes）設定すると、攻撃を検出する成功率は非常に低くなる。一方、短く（8bytes）設定した分割サイズは、合法なシステムコールと大量に重ねるため、攻撃を検出成功率は非常に低い。また、データ量の増えにつれ、誤報率も高くなる。したがって、適切な分割サイズは検出性能にとって非常に重要である。実験の結果から、16bytes と 32bytes のデータ分割は比較的適切な分割サイズである。

5.2 HashKey 生存時間とメモリ消費量

効率よく検出性能を保つため、ここではデータサイズを 16bytes に分割する。HashKey の生存時間と提案手法のメモリ消費量との関係は図５に示されている。

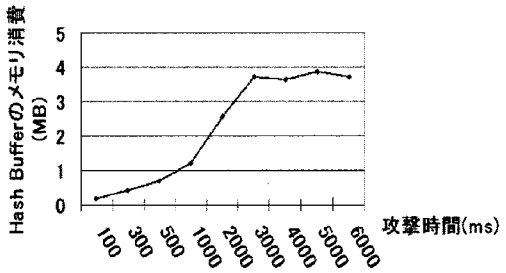

図 5. HashKey の生存時間がメモリ消費への影響

実験結果から、Hash Buffer のメモリ消費量は攻撃パケットの到着時から、急速に増えていく。HashKey の生存時間が過ぎると、メモリ消費は相対的に安定している。

5.3 Hash Buffer と検出性能

以上の実験データから、攻撃パケット個数の増えにつれ、Hash Buffer の容量も徐々に増えていき、命令キャッシュ監視ユニットによる HashKey の比較に要する時間も長くなるため、検出効率の若干な低下が見られる（図４）。HashKey の生存時間が過ぎた後、Hash Buffer の容量は安定な状態を維持していることから、Hash Buffer 容量の検出性能への影響も安定していることがわかる（図５）。

以上の分析結果を通じて、本研究の提案手法による攻撃検出率は、データの分割サイズに大きく左右することがわかった。データを16bytesごとに分割すれば、約9割の成功率で検出することが可能である。また、HashKeyに生存時間を設けることによって、急速に増えるメモリ消費の上昇を、安定的に食い止めることができる。

6．まとめと今後の課題

本研究はハードウェアレベルでのリモートバッファオーバーフロー攻撃の検出手法を提案した。また、実装モデルに対する検証実験を通じて、提案手法はリモートバッファオーバーフロー攻撃防止に有効であることが証明できた。

今回の提案はCPUの命令キャッシュを監視することによって、悪性コードの実行を防ぐことから、プロセッサアーキテクチャに依存する手法である。異なるアーキテクチャで提案手法を実装する場合は、大規模な修正が必要になる。また、本研究は、アーキテクチャレベルでの攻撃に対する有効性を検証したもので、実用化に向けては、複雑なネットワーク環境とアプリケーションの中で検証と改良は今後の課題として残されている。

参考文献

[1] Carnegie Mellon University's Computer Emergency Response Team
http://www.cert.org/

[2] T. Austin. SimpleScalar
http://www.simplescalar.com

[3] MIT Lincoln Laboratory -1999 DARPA Intrusion Detection Evaluation Data Set
http://www.ll.mit.edu/IST/ideval/data/1999/1999_data_index.html

[4] SKAPE. Understanding Windows Shellcode.
http://www.hick.org/code/skape/papers/win32-shellcode.pdf 2003-10-23.

[5] C. Cowan et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proc. USENIX Security Symp., pages 63-77, Jan. 1998.

[6] Gerardo Richarte. Four different tricks to bypass StackShield and StackGuard protection. Technical report Core Security Technologies. April 9, 2002 - June 3, 2002

[7] Fiskiran, A.M.; Lee, R.B. Runtime execution monitoring (REM) to detect and prevent malicious code execution. IEEE International Conference on Computer Design ICCD'04, 2004. 452-457