2W-02

Shadow Memoryを用いたプログラムローダにより Heap-based Buffer Overflow攻撃を緩和する手法の提案と実装

渡辺 亮平† 角田 佳史‡ 堀 洋輔‡ 馬場 隆彰‡ 宮崎 博行‡ 王 氷‡ 近藤 秀太† 齋藤 孝道†

明治大学† 明治大学大学院‡

1. はじめに

Heap-based Buffer Overflow（以下，HBoFと呼ぶ）脆弱性を悪用する攻撃に対して，OS，コンパイラ，リンカ，またはライブラリにおける様々な対策技術が考案されてきた．しかし，既存の対策技術は，様々な課題がある．

そこで，本論文では，既に配布された実行ファイルに適用できるアプリケーションプログラムローダを用いたアプローチによる対策技術の提案，実装および評価を行う．提案方式では，HBoF 脆弱性を招くライブラリ関数群は，Shadow Memoryを利用する，より安全な関数群（以下，境界検査関数群と呼ぶ）に置換される．プログラマは，提案方式のプログラムローダ上で保護対象のアプリケーションプログラム（以下，対象プログラムと呼ぶ）を起動することで，HBoF攻撃を緩和することができる．

今回想定する環境は，32ビットLinux OSおよび実行ファイルの形式は，ELFである．

2. 既存の HBoF 攻撃に対する対策技術

コンパイル時にソースコードの書き換えと，専用のライブラリをリンクすることで，ヒープ領域を含む，様々なメモリバグを検出することができる技術に AddressSanitizer[1]がある．AddressSanitizerは，Shadow Memoryというスタック領域やヒープ領域などのデータ領域のアクセス可否の情報を格納する別領域を用いて，メモリバグを検出する．Shadow Memoryの1バイトは，データ領域の8バイトに対応しており，プログラム実行時に仮想メモリ上に約512Mバイトの Shadow Memoryが割り当てられる．様々なメモリバグを検出できるが， バグを検出するためのオーバーヘッドなどの問題がある．

動的メモリ割り当て関数およびメモリ解放関数を，HBoF脆弱性を検知する関数に置換する手法がある[2][3]．これらの手法は，置換されたメモリ解放関数において，HBoF脆弱性を検知できる．しかし，メモリ解放関数以外のHBoF脆弱性を招く関数群において，HBoF脆弱性を検知することができず，HBoF攻撃を緩和できない場合がある．

Mitigation of Heap-based Buffer Overflow Attack by Program Loader Using Shadow Memory

†Ryohei WATANABE ‡Yoshifumi SUMIDA ‡Yosuke HORI ‡Takaaki BABA ‡Hiroyuki MIYAZAKI ‡Wang Bing †Shuta KONDO †Takamichi SAITO

†Meiji University ‡Graduate School of Meiji University

3. 提案方式

3.1 概要

提案方式では，対象プログラムに存在する，HBoF脆弱性を招くライブラリ関数群を境界検査関数群へ置換する．同時に，関連する動的メモリ割り当て関数群およびメモリ解放関数も，Shadow Memoryを利用する関数群に置換する．ここで，本論文で扱うHBoF脆弱性を招くライブラリ関数群および関連するライブラリ関数群を，文献[6][7]をもとに表1と2に定める．この置換を提案の一部として，プログラムローダ（以下，Safe Transローダと呼ぶ）[4][5]を用いた．

表1 HBoF脆弱性を招くライブラリ関数群

No	HBoF脆弱性を招く関数のプロトタイプ
1	char *strcpy(char *s1, const char *s2)
2	char *strncpy(char *dest, const char *src, size_t n)
3	char *stpcpy(char *dest, const char *src)
4	size_t strlcpy(char *dst, const char *src, size_t size)
5	char *strcat(char *s1, const char *s2)
6	char *strncat（char *s1, const char *s2, size_t n)
7	void *memcpy(void *buf1, const void *buf2, size_t n)
8	char *gets(char *s)
9	char *getwd（char *buf)
10	char *realpath(const char *path, char *resolved_path)
11	int sprintf(char *str, const char *format, ...)
12	int snprintf（char * restrict s, size_t n, const char * restrict format, ...)
13	int vsprintf（char * restrict s, const char * restrict format, va_list arg)
14	int vsnprintf(char * restrict s, size_t n, const char * restrict format, va_list arg)
15	int scanf(const char *format, ...)

表2 関連する動的メモリ割り当て関数群およびメモリ解放関数

No	関連する関数のプロトタイプ
1	void *malloc(size_t size)
2	void *calloc(size_t n, size_t size)
3	void *realloc(void *ptr, size_t size)
4	void free(void *ptr)

3.2 Safe Trans ローダ

Safe Trans ローダは，実行時の引数として渡された対象プログラムを仮想メモリに展開し，実行する．Safe Trans ローダは仮想メモリ（0x02000000～）に展開され，対象プログラムは，Safe Trans ローダのヒープ領域に展開される．そして，対象プログラムの ELF 情報の解析後，対象プログラムの LOAD/DYNAMIC セグメントが仮想メモリに展開される．その際に，対象プログラムに存在する置換対象のライブラリ関数群を，Safe Trans ローダに実装されている関数群に置換する．そして，対象プログラムへ制御を移す[5].

3.3 提案手法の Shadow Memory

Safe Trans ローダは，動的メモリ割り当てされたバッファサイズの情報を格納するために，AddressSanitizer 同様に，Shadow Memory を用いた．malloc 関数などの動的メモリ割り当て関数は，一般的に，少なくとも 8 バイトにアライメントされる．そこで，提案方式では，AddressSanitizer とは違い，Shadow Memory の 1 ビットに対して，動的メモリ割り当てされたバッファの 8 バイトを対応させた．32 ビット環境において，Safe Trans ローダは，実行時に，仮想メモリ上に約 48M バイトの Shadow Memory を確保する．「動的メモリ割り当てされた仮想メモリ上のアドレス」（以下，Addr と呼ぶ）から対応する Shadow Memory のアドレスは，文献[1]に倣い（Addr >> 6）+ 0x30000000 と定める．

3.4 Shadow Memory を参照する関数群

例として，安全な malloc 関数，安全な strcpy 関数，および安全な free 関数の動作について説明する（以下，それぞれ，Hmalloc 関数，Hstrcpy 関数，および Hfree 関数と呼ぶ）．

Hmalloc 関数は，次のことを行う

イ）引数のバッファサイズを元に，オリジナルの malloc 関数を実行する．

ロ）malloc 関数の戻り値であるアドレスから対応する Shadow Memory を求め，バッファサイズ分のマークを行う．

Hstrcpy 関数は，次のことを行う．

イ）書き込み先のバッファのアドレスから対応する Shadow Memory を求める．

ロ）書き込む文字列の長さと，対応する Shadow Memory のマークから，バッファを超えない書き込みであることを確認する．

ハ）超えない場合は，文字列をコピーする．そうでなければ，対象プログラムは終了する．

Hfree 関数は，次のことを行う．

Hfree 関数における HBoF 攻撃および Double Free の緩和は，Safe Trans ローダに静的リンクされた glibc 2.21 のオリジナルの free 関数によって行われる．

イ）引数のアドレスから，対応する Shadow Memory のマークを初期化する．

ロ）オリジナルの free 関数を実行する．

4 評価

4.1 HBoF 脆弱性への有効性

CWE-122[8] を記載する Mitre 社サイトに例示された HBoF 脆弱性を含むプログラム，Example 1 および Example 2 を用いて提案方式の有効性を示す．Example 1 は， strcpy 関数により文字列をコピーするプログラムであり，文字列の長さを検査せずに strcpy 関数を使用している点に HBoF 脆弱性がある． Example 2 は，引数の文字列をエスケープするプログラムであり，エスケープされた文字列の長さを検査していない点に HBoF 脆弱性がある．

結果として，Safe Trans ローダによって置換された境界検査関数群によって，Example 1 および Example 2 に含まれる HBoF 脆弱性の悪用を防ぐことができた．

4.2 オーバーヘッド

Linux 標準の ELF ローダおよび Safe Trans ローダ環境において，4.1 節で使用した Example 1 および Example 2 に 10,000,000 回ループする処理を加えたプログラム（以下，対象 1，対象 2 と呼ぶ）の実行処理時間を計測した（表 3，表 4 参照）．評価環境は，Intel(R) Xeon(R) CPU E5620@2.40GHz，および Ubuntu14.04 LTS 32bit．計測には time コマンドを使用した．

結果として，Safe Trans ローダのオーバーヘッドは，対象 1 は約 39%，対象 2 は約 10%となった．対象 1 は，Shadow Memory を利用する関数群のみをループするプログラムなので，このような結果になったと考えられる．

表3 対象1の実行処理時間

	標準ローダ	Safe Transローダ
実行処理時間	1.222[ｓ]	1.698[ｓ]（38.952%）

表4 対象2の実行処理時間

	標準ローダ	Safe Transローダ
実行処理時間	47.040[ｓ]	51.924[ｓ]（10.38%）

5 まとめ

本論文では，既に配布された実行ファイルに適用できるアプリケーションプログラムローダにより HBoF 攻撃を緩和する手法，実装，および評価を行った．

今後の課題として，メモリ破損脆弱性群の報告がされている様々な実行ファイルに対しての検証，およびベンチマークツールを用いた詳細なパフォーマンスの評価がある．

6 参考文献

[1] Konstantin,Serebryany,Derek,Bruening,Alexander,Potapenko,Dmitry, Vyukov,"AddressSanitizer: A Fast Address Sanity Checker"

[2] The GNU C Library: Heap Consistency Checking,http://www.gnu.org/software/libc/manual/html_node/Heap-Consistency-Checking.html

[3] Pageheap,http://technet.microsoft.com/jajp/library/

[4] 齋藤孝道，上原崇史，金子洋平，鈴木舞音，角田佳史，堀洋輔，馬場隆彰，宮崎博行: リリースされたバイナリに適用するスタックベース BoF 攻撃緩和技術の試作と評価

[5] 齋藤 孝道† 堀 洋輔‡ 角田 佳史‡ 馬場 隆彰‡ 宮崎 博行‡ 王 氷‡ 近藤 秀太† 渡辺 亮平†: プログラムローダにより UAF 攻撃を抑制する手法の提案と実装

[6] Robert C.Seacord 著 歌代和正，久保正樹，椎木孝斉 訳, C/C++セキュアコーディング第 2 版

[7] Building Secure Software John Viega, Gary McGraw 共著 齋藤孝道 監訳/武舎広幸・河村政雄 共役

[8] CWE-122 Heap-based Buffer Overflow https://cwe.mitre.org/data/definitions/122.html