メモリ破損攻撃への対策技術の調査と分類

齋藤 孝道‡　鈴木 舞音†　上原 崇史†　金子 洋平†　角田 佳史†　馬場 隆彰‡

†明治大学大学院，‡明治大学
214-8571　神奈川県川崎市多摩区東三田 1-1-1
saito@cs.meiji.ac.jp, {ce36028, ce36006, ce36017, ce46018, ee17033}@meiji.ac.jp

あらまし　ソフトウエアのメモリ破損脆弱性を悪用する攻撃，いわゆる，メモリ破損攻撃が次々と登場し問題となっている．メモリ破損攻撃とは，メモリ破損脆弱性のあるプログラムの制御フローを攻撃者の意図する動作に変えることである．一般的には，Buffer Overflow攻撃とも呼ばれている．OSやコンパイラでの防御・攻撃緩和機能が開発されてはいるが，それらを回避する更なる攻撃も登場している. そこで, 本論文では, メモリ破損攻撃への対策技術について調査し, 分類する.

A Survey of Prevention/Mitigation against Memory Corruption Attacks

Takamichi SAITO‡　Maine SUZUKI†　Takafumi UEHARA†　Yohei KANEKO†
Yoshifumi SUMIDA†　Takaaki BABA‡

†Graduate School of Meiji University, ‡Meiji University
1-1-1, Higashimita, Tama-ku Kawasaki-shi, Kanagawa, 214-8571, Japan
saito@cs.meiji.ac.jp, {ce36028, ce36006, ce36017, ce46029, ee17033}@meiji.ac.jp

Abstract It has become a serious problem that the number of attacks that exploits memory corruption vulnerability in software is increased. Protection/mitigation technologies against it in OS or with compilers have been developed, but further attacks have been appeared to bypass them. In this paper, we survey to classify protection or mitigation technologies against memory corruption vulnerabilities in OS or with compilers.

1　はじめに

脆弱性の種類を識別するための共通の脆弱性タイプの一覧である CWE (Common Weakness Enumeration) [1]において，メモリ破損脆弱性に分類される，いわゆる Buffer Overflow は，現在でも NVD (National Vulnerability Database) [2]での報告が絶えない脆弱性の一つである．

Buffer Overflow は，1972 年に Computer Security Technology Planning Study[3]にて初めて公に文書化された．その後，1988 年に発生した Morris Worm[4]をはじめ，2014 年に発生した Internet Explorer の Use After Free 脆弱性[5]など，現在まで様々な攻撃に悪用されている．

メモリ破損脆弱性を利用した攻撃（以降，メモリ破損攻撃という）に対して，OS やコンパイラでの防御・攻撃緩和機能（以降，対策技術という）が開発されてはいるが，様々な攻撃手法を組み合わせることによってそれらを回避する更なる攻撃も登場している．

そこで，本論文では，メモリ破損攻撃への対策技術について調査し，脆弱性に基づき分類する．また，脆弱性の分類は CWE (Version 2.8) に従う．

本論文では，主に，C 言語で書かれたプログラムかつ GCC (GNU Compiler Collection) でコンパイルされた Linux ELF (Executable and Linkable Format) 形式[6]のバイナリを 32bit の Linux OS にて利用する環境に焦点を絞る．適宜，Windows OS も比較対象として扱う．

2 Buffer Overflow 攻撃

2.1 概要

メモリ破損脆弱性 (CWE-119) [7]は，プログラム内で確保されているバッファの境界外への読み書きが可能な際に発生する．メモリ破損脆弱性を利用して，攻撃者は任意のコード実行，意図する制御フローへの書き換え，任意の情報の読み出し，または，システムの破壊が可能となる．Buffer Overflow (CWE-120) [8]とは，メモリ破損脆弱性の一種で，入力データを検査しないプログラムの脆弱性によって，プログラム内でバッファとして確保している範囲を超えて，メモリ領域にデータが書き込まれてしまう現象もしくは脆弱性のことである．この脆弱性を利用した攻撃を Buffer Overflow 攻撃という．攻撃者はバッファサイズ以上のデータでバッファを溢れさせ，プログラムの制御フローを攻撃者の意図した動作に変えるようにメモリの内容を書き換える．

2.2 Stack-based Buffer Overflow 攻撃

Stack-based Overflow (CWE-121) [9]とは，スタック領域に確保されたバッファで Buffer Overflow を引き起こす脆弱性である．スタック領域では，関数ポインタやフレームポインタ，リターンアドレスが書き換え対象となる．この脆弱性を利用した攻撃を Stack-based Buffer Overflow 攻撃という．

2.3 Heap-based Buffer Overflow 攻撃

Heap-based Overflow (CWE-122) [10]とは，ヒープ領域に確保されたバッファで Buffer Overflow を引き起こす脆弱性である．ヒープ領域では，関数へのポインタなどが書き換え対象となる．この脆弱性を利用した攻撃を Heap-based Buffer Overflow 攻撃という．

2.4 BSS-based Buffer Overflow 攻撃

BSS[11]領域とは，プログラム実行時に自動的に生成される領域で，静的変数や大域変数のうち初期化されていないものやこれらの変数に 0 が代入されているものが入る．

BSS-based Buffer Overflow とは，BSS 領域に確保されたバッファで Buffer Overflow を引き起こす脆弱性である．BSS 領域では，関数ポインタが書き換え対象となる．これに該当する CWE の分類はない．この脆弱性を利用した攻撃を BSS-based Buffer Overflow 攻撃という．

2.5 Buffer Overflow 攻撃と併用される脆弱性

Buffer Overflow 攻撃において，メモリ破損脆弱性だけでなく，書式文字列の問題 (CWE-134) や数値処理の問題 (CWE-189) を併用することで，攻撃者は，対策技術を回避することがある．ここでは，メモリ破損と併用して攻撃に利用する脆弱性について説明する．

2.5.1 書式文字列の問題

書式文字列の問題 (CWE-134, Uncontrolled Format String) [12]とは，書式指定子のない printf 系列の関数を不正利用し，メモリ上の任意の値を読み書きできてしまう脆弱性のことである．例えば，printf(input)関数呼出しが不正に使用される場合，input 変数への入力文字列が書式指定子として使用されてしまう．これを悪用する攻撃を書式文字列攻撃という．

2.5.2 数値処理の問題

数値処理の問題 (CWE-189) (Numeric Errors) [13]とは，不適切な数値演算または数値変換に関する脆弱性のことである．数値処理の問題に分類される Integer Overflow or Wraparound (CWE-190) [14]が，特に Buffer Overflow 攻撃時に併用される．

Integer Overflow or Wraparound は，演算結果の値が，演算式の型で表現できる範囲を超える場合に発生する脆弱性である．

3 既存の対策技術

ここでは，OS，コンパイラ，及び，リンカごとに，既存の対策技術を示す．

3.1 OS における対策技術

3.1.1 コード実行保護機能

コード実行保護機能とは，プログラム実行時に使用するスタック，ヒープ，.bss セクション及び.data セクション (以降，データセグメントという) ,共有ライブラリの各領域上でのコードの実行を不可能にする機能である[15][16]．GCC などのコンパイラが対象のオリジナルのプログラムをコンパイルする際，この機能の有

効／無効を指定することも可能である（後述）．

この機能は，Windows では DEP（Data Execution Prevention），Linux では Exec-Shield[19]もしくは PaX[20]と呼ばれる．DEP は，Windows XP Service Pack2 で導入された．

Windows の DEP には，プロセッサの機能である NX（NoeXecute）/XD（eXecute Disable）ビットを利用するハードウエア DEP と，利用しないソフトウエア DEP（SafeSEH）があるが，コード実行防止機能は前者にあたる[15][16]．

ここで，Linux における NX/XD ビットとは，64 ビット OS/CPU の場合，ページ・テーブル・エントリ内の MSB（Most Significant Bit）にあるが，32 ビット OS/CPU の場合は，PAE（Physical Address Extension）を適用した際のページ・テーブル・エントリ内の MSB にある．

Exec-Shield は，2002 年後半に Red Hat 社によって開発された．プロジェクト開始当初は，Linux カーネル 2.4.21-rc1 に対するセキュリティパッチとして公開されていたが，Linux カーネル 2.6.8 以降でメインストリームにマージされた．当初，Exec-Shield は，コード実行防止機能の他に，ASCII-armor（3.1.2 節参照），ASLR（3.1.3 節参照）の一部の機能が含まれていた．現在でも，RedHat 系の OS では，デフォルトで Exec-Shield の機能が利用でき，ASLR も含まれる．しかし，ASCII-armor は，PreLink 機能のオプションの一つとして機能し，Exec-Shield とは独立している．

コード実行防止機能の効果は，限定的で，例えば，Return-to-libc 攻撃[16][17]により，回避可能である．

3.1.2 ASCII-armor

ASCII-armor[19]は，RedHat により開発された Exec-Shield の機能の一つであった．現在は，Exec-Shield の機能と独立し，PreLink[21]機能のオプションのひとつとなっている．--exec-shield オプションをつけて再リンクすると，実行コードにおいて，ASCII-armor が有効になり，--no-exec-shield オプションであると ASCII-armor が無効となる．

ASCII-armor は，共有ライブラリをメモリ上に配置する際に，32 bit 版の OS では，最上位のバイトが 0x00，すなわち，0x00FFFFFF 以下のアドレスへ配置する．アドレスの上位バイトに 0x00 を挿入することで，NULL 文字（¥0x00）を終端とみなす strcpy 等によるインジェクションベクタのコピーを防止する．しかしながら，例えば，Return-to-strcpy 攻撃[22]により，ASCII-armor は回避可能である．

3.1.3 ASLR

ASLR（Address Space Layout Randomization）とは，アドレス空間配置のランダマイズ化のことで，プロセスのメモリ領域のマッピングを無作為に配置する機能である．

主に，対象領域は，プロセスのメモリ配置における実行ファイルの基底アドレス，スタック，ヒープ，及び，共有ライブラリやデータセグメント各領域の基底アドレスである．

PIE（Position-Independent Executable）形式でコンパイルした実行コードにおけるアドレスのランダマイズ機能を Full-ASLR と呼ぶ．Full-ASLR の場合は，ASLR のランダム化に加えて，テキストセグメントもランダム化する．逆に，PIE 形式でコンパイルしないバイナリの場合，OS で ASLR がサポートされていても，効果は限定的である．

Windows における ASLR は，Vista 以降に導入されているが，OS の世代ごとに適用状況が違う．Vista 以降に登場した Windows アプリケーションは，標準でコンパイル時に /DYNAMICBASE オプションが有効になっている[23]．しかし，/DYNAMICBASE オプションを有効にしていない Windows アプリケーションであっても，Microsoft EMET 4.1 の強制 ASLR を使用することで ASLR を有効化できる[24]．

Linux おける ASLR は，当初 Exec-Shield の機能の一つであったが，Linux Kernel 2.6.12（2005 年）から，メインストリームにマージされた．2000 年に初めてパッチがリリースされ，現在でも保守が続けられている PaX としても採用されている．

Linux（Kernel 3.13.0-34-generic）における ASLR のプロセスの基底アドレスのランダム化は，ローダのプログラムである binfmt_elf.c の load_elf_binary 関数で ELF ファイルが読み込まれる際に行われる．スタック領域の基底アドレスは，load_elf_binary 関数内で実行される randomize_stack_top 関数によって決められる．ヒープ領域の基底アドレスは，arch_randomize_brk 関数によって決められる．共有ライブラリがメモリマップされる領域は，setup_new_exec，arch_pick_mmap_layout，及び，mmap_base 関数が順番に実行されることで決定する．また，特に PIE 形式の際には，load_elf_binary 関数が ELF ファイルに記載されたセグメントをメモリマップする際にセグメントの配置場所を決定する．

Full-ASLR の欠点として，実行コードが PIE 形式であるので，Linux の x86 アーキテクチャではパフォーマンスの低下が生じるという問

題がある．よって，あまり普及していない．例えば，Ubuntu の場合，5~10%のパフォーマンスの低下を引き起こす．しかし，x86_64 アーキテクチャ上では，PIE 形式の顕著なパフォーマンス低下はないとされている．また，PIE 形式でコンパイルされているアプリケーションの数は，デフォルトでインストールされているアプリケーションのうち 20%未満であるので，普及しているとは言えない[25][26].

Windows と Linux の ASLR の適用の違いとしては，Windows はリンク時のオプションであるのに対し，Linux はコンパイル時のオプションであることがある．よって，Linux では，Windows の EMET の様にアプリケーションに対して強制的に ASLR を有効にできない.

ASLR の効果については，32 ビット OS の場合，特に，限定的であると言える．表 1 は，我々の実験による ASLR のエントロピー測定結果及び Linux (Kernel 3.13.0-34-generic) のソースコードに基づく理論値を示す.

表 1　ASLR のエントロピー測定結果

Dist.	カーネル	ヒープ領域のエントロピー 実測値[bit] 理論値[bit]	スタック領域のエントロピー 実測値[bit] 理論値[bit]
CentOS 6.5 32bit	Linux Kernel 2.6.32-431.el6.i686	12.94120887 13	16.42438035 20
CentOS 6.5 64bit	Linux Kernel 2.6.32-431.el6.x86_64	12.93953240 13	16.60920047 20
Ubuntu 14.04 32bit	Linux Kernel 3.13.0-34-generic	12.94038015 13	16.42608808 20
Ubuntu 14.04 32bit	Linux Kernel 3.13.0-34-generic	12.93985193 13	16.60920047 20
Mac OS X 10.9.4 64 bit	Darwin Kernel 13.3.0	8.339182292 -	15.45256796 -
Windows 7 64bit	-	6.548191779 5 ※	未採取 14 ※
Windows 8.1 32bit	-	6.374585422 8 (Windows 8)※	未採取 17 (Windows 8)※

※　Windows Server® 2012 マイグレーション ガイド[27]

3.1.4　kASLR

kASLR[28]は，カーネル空間内のプログラムやデータ配置をカーネルの起動の都度にランダムに配置する．Linux Kernel 3.14 から追加された．Linux Kernel 3.15 では，モジュール読み込み位置のランダム化を実現している．また，プロセスに関する実行情報を提供する procfs ファイルのアクセス権をプロセスの実行権限所有者のみ参照できるように変更された.

3.1.5　OS 対策機能対応状況

表 2 は，現在の Linux ディストリビューションにおける Exec-Shield，PaX 及び ASCII-Armor の対応状況を示す.

表 2　Linux の対策機能対応状況

Dist.	対応状況	保護機能
CentOS	3 以降	Exec-Shield ASCII-armor
Debian	カーネルオプションにて追加可	Exec-Shield ASCII-armor
Fedora	Core1 以降	Exec-Shield ASCII-armor PaX
Gentoo	カーネルオプションにて追加可	PaX
RedHat enterprise Linux	3 以降	Exec-Shield ASCII-armor

※Linux Kernel 2.6.12 から ASLR は標準搭載

3.2　コンパイラ／リンカの対策技術

3.2.1　GCC のコンパイラオプション

ここでは，GCC のコンパイラオプションとして，実装されている対策技術について示す.

3.2.1.1　整数オーバーフローの検出

GCC バージョン 3.4 以降では，-ftrapv オプション[29]をつけてコンパイルすると，符号付き整数同士の加減乗算における整数オーバーフローをプログラム実行時に検出可能である．整数オーバーフローを検出するとプログラムを停止する．しかし，符号なし整数同士の演算，符号付整数と符号なし整数の混合演算，符号付整数から符号なし整数への変換，及び，符号なし整数から符号付き整数への変換，ビット数の少ない型への代入による切り捨て，0 による除算は検出されないため注意が必要である．このオプションは，数値処理の問題 (CWE-189) に関する攻撃に対策として期待できる.

3.2.1.2　Mudflap

GCC バージョン 4 以降では，-g -fmudflap オプション[30]をつけてコンパイルすると，ポインタに関連する間違いをプログラム実行時に動的に検出可能である.

3.2.1.3 execstack

GCC のコンパイル時に，実行コードや共有ライブラリのコード実行保護機能の有効／無効するオプションとして，execstack (executable stack) オプション[31]がある．

コンパイル時に-z execstack オプションを引数として与えると，プログラム実行時に使用するスタック, ヒープ, データセグメント, 及び, 共有ライブラリの各領域に対するコード実行防止機能を無効にする．また，ソースコードおよび実行コードに対して，NX/ND ビットによるコード実行防止機能を有効にするには，-z noexecstack オプションをつけて，再コンパイルする必要がある．

GCC のコンパイルオプションとは別に，execstack コマンドも用意されており，以下のように使用できる．

- execstack -s [実行コード／共有ライブラリ]

実行コード／共有ライブラリのコード実行防止機能を無効にする．

- execstack -c [実行コード／共有ライブラリ]

実行コード／共有ライブラリのコード実行防止機能を有効にする．

3.2.1.4 Stack Smashing Protector

GCC で導入されている Stack Smashing Protector (以降，SSP という) は，スタック領域上に配置されるリターンアドレス，フレームポインタ，引数，ローカル変数，及び，ポインタを保護する機能である．これは，IBM の StackGuard と呼ばれる保護機能を元に開発が行われた．GCC バージョン 4.1 以前では，パッチにて公開されていたが，バージョン 4.1 以降では標準機能とされている．

SSP は，スタック上で，フレームポインタより低位に canary と呼ばれる値 (後述) を挿入し，その値が関数終了時に改竄されているかどうかを確認する．SSP には，主に，引数や関数ポインタの改竄を防ぐため，引数保護機能と関数ポインタ保護機能という機能がある[32]．これらを実現するため，実行コードの作成の際，ターゲット関数内で宣言されたバッファより低位に引数や関数ポインタを配置する．SSP を適用したバイナリの実行時，改竄を検知した場合は，実行を停止する．

現在の GCC の標準的な利用では SSP の canary は, 関数内で宣言されるバッファのサイズ 7 バイト以下だと入らず，8 バイト以上の場合に挿入される．ただし，Ubuntu10.10 以降では，4 バイト以上の場合に挿入される．

以上の標準的な SSP に加え，GCC バージョン 4.9 から-fstack-protector-strong[33]という機能が追加された．これは，それ以前からあった，SSP の処理をすべての関数に適用する -fstack-protector-all というオプションでは, 不要に適用してしまう可能性があるので，ある条件を満たす関数のみに適用するオプションである．

前述の通り，SSP 専用コードの命令列は，コンパイル時に追加する．よって，後から SSP を適用する場合はソースが必要となる欠点がある．また，Buffer Overflow の発生自体を防ぐことはできないという問題もある．更に，当然, 追加の命令列が増えるので，実行ファイルのサイズ大きくなることも懸念される．

我々は, コンパイル時に追加される SSP 専用コードの命令列によって，どの程度.text 領域が増加するのかについて，以下の環境で確認した (表 3 参照) ．Apache の場合，.text セクションが 14.26%増えていることが分かった．

計測環境

- Ubuntu 14.04 LTS 32bit
- Linux Kernel 3.13.0-29-generic
- GCC 4.9.1
- Apache httpd-2.4.9

表 3 SSP による.text セクションの肥大化状況

	①	②	③	④
.text セクション [byte]	248,146	248,626	283,583	248,626
httpd [byte]	1,564,197	1,565,245	1,605,113	1,565,269
.text セクションの割合[%]	15.86	15.88	17.66	15.88
.text セクションの肥大化[%]	-	0.193	14.26	0.193
httpd の肥大化[%]	-	0.067	2.613	0.069

① -fno-stack-protector
② -fstack-protector (デフォルト)
③ -fstack-protector-all
④ -fstack-protector-strong

SSP により，Stack-based Overflow 攻撃の多くは抑制されることが期待されてはいるが，canary 自体を偽造・上書きしたりする攻撃も発案されている．よって，その後，偽造・上書きを難しくする canary として，以下のようなものが利用されている[34].

- Random canary

実行時に/dev/random または, /dev/urandom を参照して生成した 4byte の乱数とする.

- Terminator canary

0x000AFF0D という値を利用する．NULL 文字（¥0x00）は，strcpy 関数での終端文字, 0x0A は gets 関数での改行, 0xFF は End of File (EOF)，0x0D はキャリッジリターンであるので，単純な上書きが困難になる.

- Null canary

4 バイトすべてが，NULL 文字（¥0x00）の値を利用する.

- XOR Random canary

Random canary とリターンアドレスとの XOR をとった値を利用する.

多くのディストリビューションで，GCC の SSP を利用する場合，glibc が併用されているが, GCC の SSP 機能による canary の生成は, glibc で行っている．glibc-2.19 では，Random canary の下位 1 バイトが必ず NULL 文字 (¥0x00) で残りは/dev/random による 4 バイト乱数値になる.

我々が確認したところ，Ubuntu14.04 (32 ビット）では，プロセスが生成された時点で生成された canary を使いまわしているということがわかった．よって，たとえば，親プロセスで生成された canary を何らかの方法で取得できれば，子プロセスでは，canary を偽造できる可能性があることが分かる[34].

3.2.1.5 Automatic Fortification

Automatic Fortification は，GCC バージョン 4.0 から標準で導入されたセキュリティ機能で，Buffer Overflow 攻撃から実行コードを保護する．これは，Buffer Overflow の危険性を持つ（共有ライブラリにある）関数群を安全な関数に置換する機能である．コンパイル及び実行時に，バッファ領域を超えるサイズの文字列が渡されていないかのチェックを行う．リターンアドレスやフレームポインタの改竄を未然に防ぐ．ここで，Automatic Fortification の類似の技術として, 動的リンカを利用する libsafe [35]がある．これは，strcpy 関数などの Buffer Overflow を引き起こしやすい関数が呼ばれる前に制御をフックし，コピー先のバッファが十分確保されているかどうかをコピー前にチェックする.

GCC によるコンパイル時の-O1 以上の最適化と D_FORTIFY_SOURCE=2 というオプションを加えることにより，実行時に，書式文字列攻撃（CWE-134）を検出する機能も備えている．ただし，ある関数で宣言されているバッファを引数として別の関数に渡し，その別の関数内で引数として渡されたバッファに対して，何かしらの文字列をコピーする際には，この機能は働かない.

3.2.1.6 RELRO

RELRO（RELocation Read-Only）とは，ELF 形式のバイナリにおける各セクション, 主にライブラリ関数のアドレスを保持する GOT (Global Offset Table) 領域を読み取り専用にする動的リンカのセキュリティ機能である.

遅延バインド有効の際の RELRO である Partial RELRO の場合には, 読み書き可能である.got.plt セクションが生成される. したがって, Partial RELRO の場合には，GOT 書き換え攻撃が行われる可能性がある[36].

遅延バインド無効の際の Full RELRO の場合には，.got.plt セクションは生成されず，データセグメント以外読み取り専用となる.

3.2.2 Stack-Shield

2000 年に提案された Stack-Shield[37]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コンパイラの対策技術である．GCC のパッチという形で提供されている[38]．関数終了直前にリターンアドレスのコピーを強制的にスタックに上書きする．関数終了時に上書きされたリターンアドレスに実行が移る.

3.2.3 その他の対策技術

ISR (Instruction Set Randomization) [39]とは，既存のバイナリコードを等価な命令で置き換えることで ROP 攻撃を防ぐことを期待する技術であるが，普及していない.

Pin[40]を用いて，実行時に，実行コードを書き変えて, RAD (Return Address Defense) [41]を埋め込む対策技術が提案されている[42]．ただし，パフォーマンスなどの観点から，課題も多い．ここで，Pin とは，プログラムの実行時にバイナリの改変を行えるソフトウエアである．プログラムを実行する際，利用者の作成した命令コードを挿入することで，プログラムの解析や Binary Translation[43]を行うことが可能である．Pin は，2005 年には一般にリリースされている．2013 年には，IA-32，x86-64 や，Intel Xeon Phi のアーキテクチャに対応している．Linux，Windows，及び，OS X 上で利用可能である.

4　考察

4.1　OS 対策

kASLR などの新たな対策技術が 2013 年から追加されるなど，日々進歩している．比較的新しい OS (Linux Kernel 2.6.12 以降) では，デフォルトで ASLR が機能している．しかし，特に 32 ビット OS では，ASLR のエントロピーは低いため，何回かの実行に一度は同じ基底アドレスになってしまうという欠点がある．状況によっては補助的な位置づけであると言える．

4.2　コンパイラ・リンカ対策

比較的新しいバージョンのコンパイラやリンカでは，デフォルトで Buffer Overflow 対策が機能している．しかし，適用されていない場合や，より機能を強化するためには，コンパイルオプションやリンカオプションを追加し，再コンパイルまたは，再ビルドを行わなければならない．また，効率化を図るためにデフォルトでは弱いオプションで動作していることが多く，攻撃の余地を残しているといえる．

ディストリビューションによって，デフォルトで動作している対策技術がそれぞれ異なっているため注意が必要である．

4.3　対策技術と攻撃の対応

ここで，表より代表的な対策技術と攻撃の対応を行う（表 4 参照）．

表 4　対策技術と代表的な攻撃の対応表

	Heap BoF	ret2libc	Ret2strcpy	ROP	ret2reg	GOT	SOP
Exec-Shield	×	○	○	○	○	○	○
ASCII-armor	N/A	△	○	○	○	△	○
ASLR	×	×	×	△	○	×	△
SSP	N/A	×	×	×	×	△	○
RELRO	N/A	N/A	N/A	○	N/A	×	N/A

- Heap BoF (Heap-based Buffer Overflow 攻撃)
- ret2libc (Return-to-libc 攻撃)
- ret2strcpy (Return-to-plt 攻撃)
- ROP (Return-Oriented Programming 攻撃)
- ret2reg (Return-to-Register 攻撃)
- GOT (GOT 書き換え攻撃)
- SOP (String-Oriented Programming 攻撃)

5　まとめ

本論文では，メモリ破損攻撃に対する対策技術に関する調査と分類を行った．現在でも様々な対策機能が存在するが，様々な攻撃手法を組み合わせることにより，それらを回避する新たな攻撃手法も現れている．今後，対策技術の改善，新たな対策技術の考案，及び効果的な組み合わせを考えていく必要があると考えられる．本論文で述べたその他の対策技術についても引き続き調査を進めていきたい．

参考文献

[1] CWE: Common Weakness Enumeration, http://cwe.mitre.org/index.html

[2] NVD: National Vulnerability Database, http://nvd.nist.gov/

[3] NIST "Computer Security Technology Planning Study", http://csrc.nist.gov/publications/history/ande72.pdf

[4] Morris Worm, http://ja.wikipedia.org/wiki/Morris_worm

[5] Vulnerability Summary for CVE-2014-1776, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1776

[6] The Linux ELF HOWTO, http://cs.mipt.ru/docs/comp/eng/os/linux/howto/howto_english/elf/elf-howto.html#toc1

[7] CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer, http://cwe.mitre.org/data/definitions/119.html

[8] CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'), http://cwe.mitre.org/data/definitions/120.html

[9] CWE-121: Stack-based Buffer Overflow, http://cwe.mitre.org/data/definitions/121.html

[10] CWE-122: Heap-based Buffer Overflow, http://cwe.mitre.org/data/definitions/122.html

[11] Stevens, W. Richard (1992). Advanced Programming in the Unix Environment. Addison–Wesley. Section 7.6., http://bluetechs.files.wordpress.com/2014/03/advanced-programming-in-the-unix-environment-by-w-richard-stevens-stephen-a-rago-ii-edition.pdf

[12] CWE-134: Uncontrolled Format String, http://cwe.mitre.org/data/definitions/134.html

[13] CWE-189: Numeric Errors, http://cwe.mitre.org/data/definitions/189.html

[14] CWE-190: Integer Overflow or Wraparound, http://cwe.mitre.org/data/definitions/190.html

[15] 角田 佳史, 金子 洋平, 鈴木 舞音, 上原 崇史, 齋藤 孝道, バッファオーバーフロー攻撃に関する防御技術の調査, 2014 年, FIT 情報科学技術フォーラム

[16] 齋藤孝道，マスタリング TCP/IP 情報セキュリティ編，オーム社，2013

[17] Infosec Writers,"Bypassing non-executable-stack during exploitation using return-to-libc", http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf

[18] Limiting buffer overflows with ExecShield, http://www.redhat.com/magazine/009jul05/features/execshield/

[19] "Exec Shield", new Linux security feature, http://lwn.net/Articles/31032/

[20] PaX, http://pax.grsecurity.net/

[21] ELF prelink - Ubuntu Manpage Repository, http://manpages.ubuntu.com/manpages/precise/man8/prelink.8.html

[22] Linux exploit development part 4 - ASCII armor bypass + return-to-plt, http://ihazomgsecurityskillz.blogspot.jp/2011/05/linux-exploit-development-part-4-ascii.html

[23] /DYNAMICBASE (ASLR (Address Space Layout

Randomization）の使用),
http://msdn.microsoft.com/ja-jp/library/bb384887.aspx
[24] Enhanced Mitigation Experience Toolkit 4.1, file:///C:/Users/maine/Downloads/EMET_Users%20Guide_J_4.1.pdf
[25] Payer, Mathias. "Too much PIE is bad for performance." (2012).
[26] Differences Between ASLR on Windows and Linux, http://www.cert.org/blogs/certcc/post.cfm?EntryID=191
[27] Windows Server 2012 の標準機能, http://download.microsoft.com/download/0/4/4/04402DFB-65F7-435E-8861-E64A8FCAB271/WIN2012MigWP.docx
[28] Kernel address space layout randomization, http://lwn.net/Articles/569635/
[29] Options for Code Generation Conventions, https://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/Code-Gen-Options.html
[30] Mudflap Pointer Debugging , https://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging
[31] man execstack, http://pwet.fr/man/linux/administration_systeme/execstack
[32] オープンソース・ソフトウェアのセキュリティ確保に関する調査報告書, 第Ⅲ部, セキュアな実行コードの生成・実行環境技術に関する調査, https://www.ipa.go.jp/files/000013695.pdf
[33] -fstack-protector-strong, http://www.outflux.net/blog/archives/2014/01/27/fstack-protector-strong/
[34] Seredinschi, Dragoş-Adrian, and Adrian Sterca. "ENHANCING THE STACK SMASHING PROTECTION IN THE GCC." Studia Universitatis Babes-Bolyai, Informatica 56.4 (2011)
[35] libsafe, http://directory.fsf.org/wiki/Libsafe
[36] Müller, Tilo. "ASLR smack & laugh reference." Seminar on Advanced Exploitation Techniques. 2008.
[37] A stack smashing technique protection tool for Linux, http://www/angelre.com/sk/stackshield
[38] Stack Shield A "stack smashing" technique protection tool for Linux, http://www.angelfire.com/sk/stackshield/download.html
[39] Kc, Gaurav S., Angelos D. Keromytis, and Vassilis Prevelakis. "Countering code-injection attacks with instruction-set randomization." Proceedings of the 10th ACM conference on Computer and communications security. ACM, 2003.
[40] Pin - A Dynamic Binary Instrumentation Tool, https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[41] Tzi-cker Chiueh and Fu-hau Hsu, ”RAD: A compile time solution for buffer overflow attacks”, ICDCS 2001
[42] Protection against Buffer Overflow Attacks via Dynamic Binary Translation, Reliable and Autonomous Computational Science 2010
[43] binarytranslator, http://www.binarytranslator.com/