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

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†

†明治大学，‡明治大学大学院

あらまし ソフトウエアの脆弱性を悪用する攻撃において，ヒープ領域内のメモリブロック解放に伴うダングリングポインタを悪用するUse After Free（UAF）脆弱性を悪用した攻撃が2006年に登場した．近年，UAF脆弱性を悪用した攻撃は，ブラウザなどの一般的に普及しているソフトウエアに多くの報告があり，問題となっている．そこで，本論文では，プログラムローダによるUAF攻撃を抑制する手法を提案し実装する．また，その評価を行う．

Safe Trans Loader: Mitigating UAF Attack by Program Loader

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

†Meiji University， ‡Graduate School of Meiji University

Abstract In the area of software security, the attack exploited Use After Free (UAF) vulnerabilities was reported in 2006. When the target vulnerable application runs, the attack exploits a dangling pointer after the heap memory is released. Until today, it has been frequently reported that the UAF attack is found in the popular software such as a browser. It is a serious problem in the software security. In the paper, we propose to implement the application-level program loader to mitigate the UAF attack, and evaluate it.

1 はじめに

ソフトウエアの共通脆弱性タイプ一覧CWE（Common Weakness Enumeration）[1]に CWE-416[2]として分類される Use After Free（以降，UAF と呼ぶ）脆弱性が2006 年に初めて報告された．UAF 脆弱性の報告件数は，NVD（National Vulnerability Database）[3]によると，2010 年から増加し始め，現在でも一定数の報告がある（図 1 参照）．

UAF 脆弱性は，主に C や C++のメモリ解放に伴う脆弱性だが，Internet Explorer（IE）や Google Chrome などの主要なブラウザにおける JavaScript や Adobe 社の Flash Player などの一般的に普及しているプログラムにおいても報告されている[4].

UAF 脆弱性を悪用する攻撃（以降，UAF 攻撃と呼ぶ）は，攻撃対象のソフトウエアの実行中において，ヒープ領域内のメモリブロック解放後に起きる不正なポインタであるダングリングポインタを，攻撃者が悪用することによって，シェルコードに制御を移すことを目的とする．

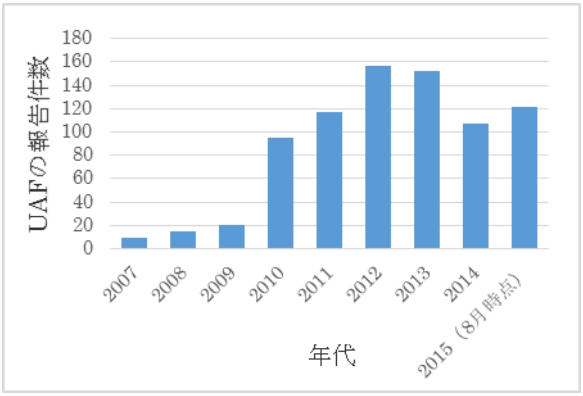

図 1:NVD における UAF 脆弱性の報告件数

脆弱性が報告される一方で，UAF 攻撃を防止および緩和する手法が様々提案されてきている．しかし，既存の UAF 攻撃を防止および緩和する手法には，プログラム実行中に UAF 攻撃を防止および緩和できないことがある，もしくは，ライブラリを書き換える必要があるなどの問題点がいくつか指摘されている．

そこで，本論文では，OS におけるプログラムローダとは別にアプリケーションとしてのプログラムローダを用いて，プログラムのロード時に，free 関数をより安全な関数に置き換えることにより，UAF 攻撃を緩和する手法の提案と実装を行う．

2　関連知識

2.1 UAF 脆弱性について

UAF（CWE-416）とは，ヒープ領域において，一度解放したメモリブロックをその解放されたメモリブロックを指し示していたポインタが再度利用されてしまう現象および脆弱性である．UAF 脆弱性によって，実行中のプログラムは，予期せぬクラッシュやダングリングポインタを引き起こす．ここで，ダングリングポインタとは，メモリブロック解放後に，解放されたメモリブロックを指し示したままの状態のポインタのことである．UAF 攻撃では，UAF 脆弱性によって発生したダングリングポインタを攻撃者が悪用し，攻撃者の用意するシェルコードを実行させる．

2.2 ELF について

ELF（Executable and Linking Format）形式[5]は実行ファイル形式の一つであり，Linux OS などで採用されている．

ELF 形式ではセクションおよびセグメントと呼ばれる単位でファイル情報が管理されている（図 2 参照）．セクションはリンカで扱われ，セグメントはローダで扱う単位である．ローダは ELF の情報を元に，ロード対象プログラムの実行直前に，各種セグメント単位で仮想メモリ上に展開する．

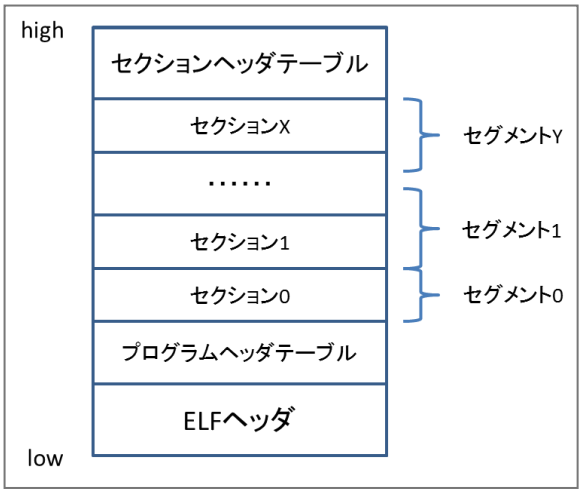

図 2:ELF の構造

セクションはプログラムを実行するために必要な各種情報を管理している．例として，プログラムの命令コードが格納されている.text セクションや読み込み専用のデータを格納する.rodata などのセクションがある．他方，セグメントは類似したセクションをまとめて管理している単位である．例として，.text セクションや.data セクションなどのメモリをマップするセクションをまとめた LOAD セグメントや.dynamic セクションなどの動的リンク情報に関するセクションをまとめた DYNAMIC セグメントなどのセグメ

ントがある．

プログラム実行時，共有ライブラリを動的にリンクする際に利用されるセクションとして，.got，.got.plt および.plt セクションがある．.got および.got.plt セクションは動的リンクされる共有ライブラリの関数のアドレスを格納するセクションである．.plt セクションは.got.plt や.got セクションに格納されている関数のアドレスを呼び出す際に扱われるセクションである．

ここで，例として，.got.plt セクションを用いて共有ライブラリの free 関数の呼び出しを説明する（図 3 参照）．.text セクションから free 関数の呼び出し際，.plt セクション内の free@plt が呼ばれる．その後，.got.plt セクションを参照して，アドレス解決済みであれば free 関数の実体を呼び出す．一方，アドレス解決前であれば，.dynamic セクションによって共有ライブラリ内の free 関数のアドレスが計算され，.got.plt セクションに格納し，アドレス解決済みの状態にする[6]．

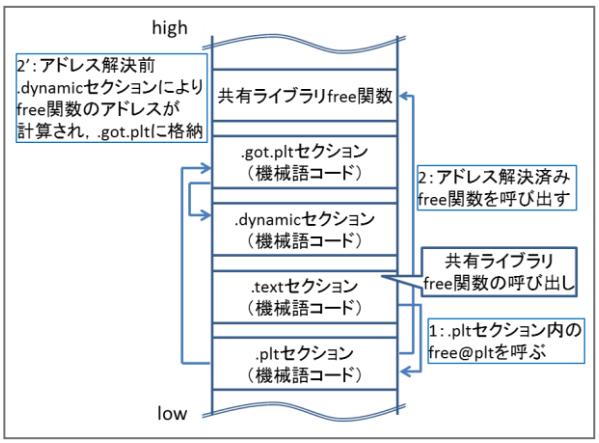

図 3:.got.plt セクションを用いた
共有ライブラリの呼び出し

3　既存方式の問題点

現在までに，UAF 攻撃を防止および緩和する手法が多く提案されている．文献[7]では，既存の UAF 攻撃の防止および緩和技術の特徴および問題点が考察されている．これをもとに，本論文での論点を整理する．

UAF 攻撃の起点として悪用されるダングリングポインタを検出するため，プログラム実行前に解析を行う手法がある．手法の種類は，動的バイナリ変換，および，テイント解析またはバイナリを静的に解析する手法などがある[8][9][10][11][12][13][14][15]．この手法は，UAF 脆弱性を残存した場合，プログラム実行中の UAF 攻撃を防ぐことができない可能性がある．

プログラム実行中にダングリングポインタを検出する技術として，コンパイル時に UAF 攻撃を検出するコードを追加する手法がある[13][16]．この手法は，プログラム実行中に UAF 攻撃を検知することで，UAF 攻撃を防ぐ．ただし，すでにコンパイル済みのプログラムに対しては適用できない問題点がある．

既存のメモリ確保および解放処理のプロセスに変更を加えたライブラリへ置き換えることで UAF 攻撃を防ぐ手法がある．この手法には，ページ単位で割り当てる malloc に置き換える手法[17][18][19]や，再利用するメモリブロックに条件を付与することで制限する手法[7][20]といった実現法がある．プログラム実行中に UAF 攻撃を防ぐことができるが，ライブラリを置き換えることが必要である点やメモリ使用効率の問題がある．

セキュリティパッチを適用するなどによって，保護対象のアプリケーションプログラムの改変することで，UAF 攻撃を防ぐ手法がある．例として，IE のメモリ破損の脆弱性 CVE-2014-1770 を修正する MS14-035 [21]，IE のメモリ破損の脆弱性 CVE-2014-1763 及び CVE-2014-1765 を修正する MS14-037 [22]の 2 つのセキュリティパッチがある．

MS14-035 では，ヒープ領域を用いる関数がメモリブロックを確保する際に IE 専用のヒープ領域から割り当てる対策がとられ，MS14-037 では，メモリブロック解放時の処理を遅延させる対策がとられた．これらは，IE のみを UAF 攻撃から保護するに留まる．つまり，ほかのプログラムに適応できない．

4　提案方式

4.1 概要

UAF 攻撃において, 再利用されるメモリブロックは解放された直後のメモリブロックが多いことが文献[7][22]において示された．そこで，本論文での提案方式では，プログラムローダを用いて，保護対象とするアプリケーションプログラム（以降，対象プログラムと呼ぶ）上で利用されているオリジナルの free 関数を置き換え，解放処理を行うタイミングを遅延させる独自の free 関数（以降，Hfree 関数と呼ぶ)に置き換える. この置き換えを，提案方式の一部として用意するプログラムローダ（以降，Safe Trans ローダと呼ぶ）により，対象プログラムをロードする際に行う．これにより，ライブラリや対象プログラムの改変および再コンパイルせずに，プログラム実行において UAF 攻撃の対策ができる．

本論文では，文献[23]のプログラムローダを改変し, Safe Trans ローダとした. ここで，対象プログラムは，Linux 環境で採用されている ELF 形式である．紙面の都合上詳細を省くが，Safe Trans ローダにより，C 言語における strcpy 関数など, いわゆる安全でない関数群[25]を，安全な関数群に置き換えることも可能である．その一つの例として，UAF 攻撃の対策としての適用である．

4.2 提案方式の詳細

(ア)　Safe Trans ローダ

Safe Trans ローダの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（0x20000000~）と対象プログラム（0x08040800~）は，同一の仮想メモリ上のヒープ領域に展開する（図 4 参照）．

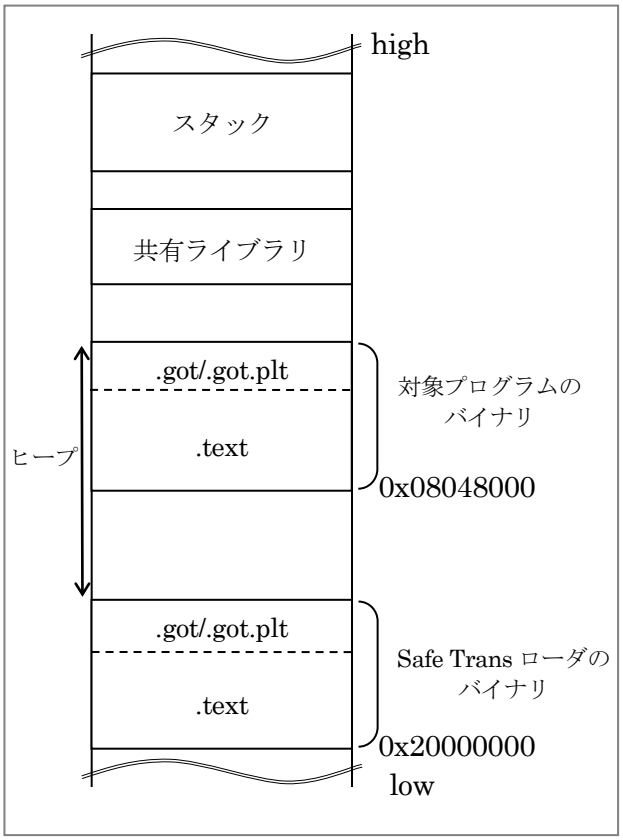

図 4: Safe Trans ローダが，対象プログラムを展開した際の仮想メモリレイアウトの概念図

Safe Trans ローダは，メモリ展開に先立ち，対象プログラムの ELF ヘッダ，プログラムヘッダ，またはセクションヘッダを解析する．その結果，各セグメントや各セクションの開始位置，オフセット，サイズ，または属性などの情報を取得する．

Safe Trans ローダにおいて, 対象プログラムの LOAD および DYNAMIC セグメントの扱いが重要となる．対象プログラムの DYNAMIC セグメントに含まれる共有ライブラリの関数のアドレスは，対象プログラムの DYNAMIC セグメントが仮想メモリ上に展開された後, 適切なアドレスに置換される．具体的には，対象プログラムの.got および.got.plt セクションを含む DYNAMIC セグメントが仮想メモリ上に展開された後，先に

示した 2 つのセクションにおける（オリジナルの）free 関数のアドレスを，Safe Trans ローダで宣言されている対応する Hfree 関数のアドレスに置換する．ただし，事前に共有ライブラリの関数のアドレスが解決されている場合は，そのアドレスに置換する．

Safe Trans ローダにより，オリジナルの free 関数の置換が行われるセクションは，図 4 における「対象プログラムのバイナリ」内の.got および.got.plt セクションである. Safe Trans ローダでは，対象プログラムのロードの際, .got および.got.plt セクションにおける共有ライブラリの関数のアドレスを事前に置換することによって，安全でないオリジナルの free 関数からより安全な Hfree 関数への置換を実現している．ここで，Hfree 関数は，Safe Trans ローダ内でグローバルな関数として宣言される．

Safe Trans ローダは，対象プログラムにおける LOAD および DYNAMIC セグメントのみを仮想メモリ上の適切な開始位置から展開し，他のセグメントは Safe Trans ローダおよび対象プログラムと共有している．

対象プログラムのロードを終えたら，Safe Trans ローダから対象プログラムへ実行を移すために，プログラムカウンタを対象プログラムの.text セクションの先頭に設定する．

```
1  void* Hfree(void *ptr){
2
3    /* 初期化 */
4    static int f_bottom = 0;
5    static int f_count = 0;
6    static void *free_queues[7] = {NULL};
7    static int f_flag = 0;
8
9    /* 解放処理 */
10   if(f_flag){
11     free(free_queues[f_bottom]);
12     f_bottom = (f_bottom+1)%7;
13   }
14
15   /* free()に渡されたポインタを確保 */
16   free_queues[f_count] = ptr;
17   if(f_count==6){
18     f_flag = 1;
19   }
20   f_count = (f_count+1)%7;
21 }
```

図 5:Hfree 関数のコード

(イ)　Hfree 関数

図 5 に，安全なメモリ解放を実現する Hfree 関数を示す．

Hfree 関数では, 対象プログラムの実行中, free 関数が呼び出されるたびに，解放する予定のメモリブロックを指すポインタを Safe Trans ローダにて static に宣言されているキュー（free_queue[]）に保存する．キューの深さ（今回の実験では 7 とした）と同じ回数だけ, 対象プログラムの実行中に free 関数が呼び出された時, メモリブロックを解放する．メモリブロック解放処理は，キューに確保されているポインタ（free_queues[f_bottom]）を保存の古い順で一つ解放する（図 6 参照）．

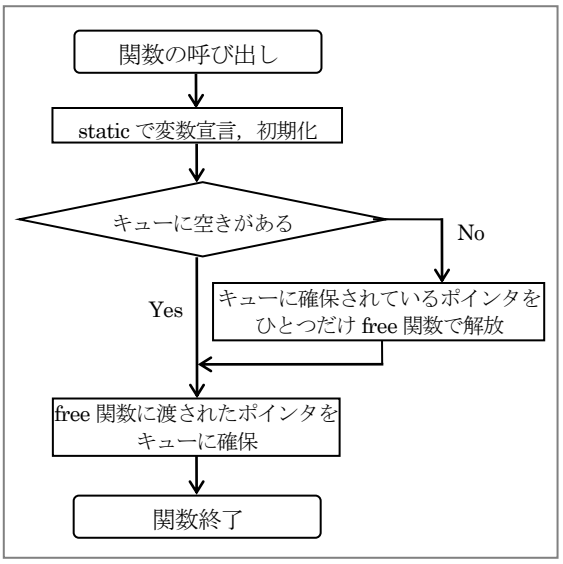

図 6:Hfree 関数のフローチャート

5　評価

5.1　UAF 攻撃への有効性

本論文では，UAF 脆弱性を含むプログラム[24]を参考に対象プログラム（以降，対象 X と呼ぶ）を用意し（図 7 参照），提案方式の有効性を確認した．Safe Trans ローダにより，対象 X に対する UAF 攻撃を以下に示す

通り再現したところ，UAF 攻撃を防いだ．

開発および評価環境として，Ubuntu 12.04.5 LTS 32bit，及び，コンパイラには gcc 4.6.3 を用いた．また，ASLR 機能[25]とデータ領域におけるコード実行防止機能[25]を無効にした状態で行った．

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  struct Str {
6      int flag;
7      char *buf;
8  };
9
10 int main(int argc, char *argv[])
11 {
12     struct Str *str;
13     int size;
14     char *buf2;
15
16
17     str = malloc(sizeof(struct Str));
18     str->flag = 1;
19     str->buf = malloc(100);
20
21     strncpy(str->buf, argv[1], 100);
22
23     free(str->buf);
24     free(str);
25
26     size = atoi(argv[2]);
27     buf2 = malloc(size);
28
29     strncpy(buf2, argv[3], size);
30
31     strncpy(str->buf, argv[4], 100);
32
33     free(buf2);
34     return 0;
35 }
```

図 7:UAF 脆弱性を含むコード

対象 X では，まず，構造体 str と構造体メンバー変数 buf をヒープ領域にメモリブロック確保する．このとき，str に構造体メンバー変数 flag 及び buf へのポインタの計 8byte が格納され，buf は 100byte のメモリブロックを指す状態になる．その後，buf にシェルコード(argv[1]により与えられる)を格納し，buf と str が指し示すメモリブロックを解放する．

次に，変数 buf2 はヒープ領域にメモリブロックを確保する．この際，str が指していたメモリブロックを再利用させるために，buf2 に与えるメモリブロックのサイズは 8byte（argv[2] により与えられる）とする．その後，buf2 への書き込みにおいて，buf のポインタが格納されていたメモリブロックに.got および.got.plt セクションにある free 関数へのアドレスを指し示すアドレス（argv[3] により与えられる）を書き込む．

31 行目において，UAF 脆弱性があり，すでに解放された buf へのポインタが格納されていたアドレスから書き込みが行われる．この際, 書き込み先は.got 及び.got.plt セクションにある free 関数を指し示すアドレスであるので, .got および.got.plt セクションにある free 関数へのアドレスをシェルコードが格納された buf へのポインタ（argv[4] により与えられる）に書き換える．

以上により，33 行目の free 関数の呼び出しの際，実行が free 関数ではなく，buf に格納されているシェルコードに移り，シェルが立ち上がる．

5.2 オーバーヘッド

本論文では，Linux の標準ローダおよび Safe Trans ローダ環境において，対象 X を 1,000,000 回実行した時の実行処理時間を計測した（表 1 参照）．評価環境として，Intel

表 1: 対象 X を 1,000,000 回実行した時の実行処理時間

	Linux の標準ローダ	Safe Trans ローダ
real（標準ローダからの増加率）	75m50.449s（0%）	78m50.993s（3.968%）
user（標準ローダからの増加率）	1m2.524s（0%）	1m9.344s（10.908%）
sys （標準ローダからの増加率）	34m44.128s（0%）	35m15.248s（1.493%）

社 Core(TM) i5-3450 CPU @ 3.10GHz において Ubuntu 12.04.5 LTS 32bit を利用した．コンパイラには gcc 4.6.3 を用いた．time コマンドを用いて計測した時間は，プログラムの呼び出しから終了までにかかった実時間（real)，プログラム自体の処理時間（user)，プログラムを処理するために、OS が処理をした時間（sys）の 3 つである．

結果として，Safe Trans ローダ環境上の実行処理時間は，標準ローダの処理時間の約 4%増しとなった．

6 考察

Safe Trans ローダが UAF 攻撃に有効であることが分かった．しかし，Safe Trans ローダには，メモリ使用効率の問題がある．malloc 関数に渡されているメモリブロックのサイズに依存して，キュー全体が確保しているメモリブロックのサイズが増加する可能性がある．その対策として，キューに確保されているメモリブロックの合計サイズに応じてメモリブロックを解放する閾値を設定するなどの更なる工夫が必要である．

今回の実装では, UAF 攻撃への対策として, free 関数を対象としたが，安全でない関数を Safe Trans ローダ内でグローバル関数として宣言し，安全でない様々な関数を安全な関数に置換することができるので，たとえば，プログラマが配慮なしコーディングしたとしても，C11 や ISO/IECTR24731-2:2010 で定義されている，より安全な関数に置き換えるという対策を併用できる．

7 おわりに

本論文では，UAF 攻撃を緩和する Safe Trans ローダを提案し実装した．Safe Trans ローダは，C 言語における strcpy 関数など，いわゆる安全でない関数群を，安全な関数群に置き換えることも可能である. 本論文では，その一つの適用例として, UAF 攻撃の対策を示した．

今後の課題として, 様々な事例での有効性の確認と，パフォーマンスに関する評価を行うことがある．

参考文献

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

[2] CWE-416: Use After Free, https://cwe.mitre.org/data/definitions/416.html

[3] NVD: National Vulnerability Database, https://nvd.nist.gov/

[4] マイクロソフトインテリジェンスレポート第 16 版 （2013 年 12 月），http://www.microsoft.com/ja-JP/download/details.aspx?id=42646

[5] John R. Levine 著（1999），Linker & Loader，Morgan Kaufmann

[6] 七誌の開発日記：ELF の動的リンク(1) http://7shi.hateblo.jp/entry/2013/05/25/103050

[7] 山内 利宏，池上 祐太：メモリ再利用禁止による Use-After-Free 脆弱性攻撃防止手法の実現と評価, Information Processing Society of Japan 2015 情報処理学会報告書

[8] Serebryany, K., Bruening, D., Potapenko, A. and Vyukov, D.: Addresssanitizer: A fast address sanity checker, Proc. 21th USENIX Conference on Annual Technical Conference, pp.309–318 （2012）

[9] Caballero, J., Grieco, G., Marron, M. and Nappa, A.: Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities, Proc. 21th International Symposium on Software Testing and Analysis, pp.133–143 （2012）

[10] Nethercote, N. and Seward, J.: Valgrind: A framework for heavyweight dynamic binary

instrumentation, Proc 11th ACM SIGPLAN Conference on Programming Language Design and Implementation）, pp.89–100 （2007）
[11] Bruening, D. and Zhao, Q.:
Practical Memory Checking with Dr. Memory, Proc. 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 213–223,（2011）
[12] Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L. and Lee, W.:
Preventing Use-after-free with Dangling Pointers Nullification, Proc. 22th Annual Network and Distributed System Security Symposium, NDSS （2015）
[13] Bruening, D. and Zhao, Q.:
Safedispatch: Securing C++ Virtual Calls from Memory Corruption Attacks, Proc. 21th Network and Distributed System Security Symposium,pp.1–15 （2014）
[14] Potet, M.L., Feist, J., Mounier, L.:
Statically Detecting Use After Free on Binary Code, Journal of Computer Virology and Hacking Techniques, Vol.10, No.3, pp.211–217 （2014）
[15] Dewey, D. and Giffin, T.:
Static Detection of C++ Vtable Escape Vulnerabilities in Binary Code, Proc. 19th Network and Distributed System Security Symposium,pp.1–14 （2012）
[16] Eigler, C.F.:
Mudflap: Pointer Use Checking for C/C++,
http://gcc.fyxm.net/summit/2003/mudflap.pdf
[17] Pageheap
http://technet.microsoft.com/ja-jp/library/cc835607.aspx
[18] Electric fence,
http://elinux.org/Electric_Fence
[19] DUMA,
http://duma.sourceforge.net/
[20] Akritidis, P.:
Cling: A Memory Allocatorto Mitigate Dangling Pointers, Proc. 19th USENIX Conference on Security, pp.177–192 （2010）
[21] Tang, J.:
Isolated heap for inter-net explorer helpsmitigate uaf exploits,
http://blog.trendmicro.com/trendlabs-security-intelligence/isolated-heap-for-internet-explorer-helps-mitigate-uaf-exploits/
[22] Tang, J.:
Mitigating uaf exploits with delay freefor internet explorer,
http://blog.trendmicro.com/trendlabs-security-intelligence/mitigating-uaf-exploits-with-delay-free-for-internet-explorer/
[23] 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
Symposium on Cryptography and Information Security 2015
[24] ももいろテクノロジー，use-after-free による GOT overwrite をやってみる，
http://inaz2.hatenablog.com/entry/2014/06/18/215452
[25] 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.