メモリ再利用を禁止するライブラリにより Use-After-Free 脆弱性攻撃を防止する手法の提案

池上 祐太　　山内 利宏

岡山大学大学院自然科学研究科
700-8530 岡山県岡山市北区津島中 3-1-1
ikegami@swlab.cs.okayama-u.ac.jp，yamauchi@cs.okayama-u.ac.jp

あらまし 近年，脆弱性攻撃において，解放後のメモリ領域を参照するダングリングポインタを悪用した Use-After-Free 脆弱性攻撃が増加している．特に，ブラウザのような大規模のプログラムは，ダングリングポインタが多く存在し，Drive-by download 攻撃で Use-After-Free 脆弱性が頻繁に利用されている．本稿では，ライブラリを改変することで，Use-After-Free 脆弱性攻撃を防止する手法を提案する．提案手法は，保護対象のプログラムを改変せずに適用でき，解放されたメモリ領域の再利用を一定期間禁止することで，Use-After-Free 脆弱性攻撃を防止できる．

Use-After-Free Prevention Method using Memory Reuse Prohibited Library

Yuta Ikegami　　Toshihiro Yamauchi

Graduate School of Natural Science and Technology, Okayama University
3-1-1, Tsushima-naka, Kita-ku, Okayama, 700-8530, JAPAN

Abstract Recently, there is an exploit that attackers use Use-After-Free vulnerability, which abused a dangling pointer that referring to a freed memory has been increasing. Particularly, large scale programs such as the browser often include a lot of dangling pointers and the Use-After-Free vulnerability is used by Drive-by download attack frequently. This paper proposes Use-After-Free prevention method using memory reuse prohibited library. Because this library prohibits freed memory area from being reused during the prescribed period, the proposed method can prevent Use-After-Free.

1 はじめに

近年，脆弱性攻撃において，解放後のメモリ領域を参照するダングリングポインタを悪用し，任意のコードを実行する Use-After-Free 脆弱性攻撃 (以降，UAF 攻撃と略す) が増加している．文献 [1] を基に，2006 年から 2014 年までの UAF 脆弱性の調査結果を図 1 に示す．2011 年以降，UAF 脆弱性の発見数は，急増している．特に，ブラウザのような大規模のプログラムは，ダングリングポインタが多く存在し，Drive-by download 攻撃で UAF 攻撃が頻繁に利用されている．また，多くのブラウザは，内部に JavaScript エンジンを持つ．このため，攻撃者は，ある程度自由にメモリを確保・解放できる JavaScript を利用し，UAF 攻撃を達成する．

これらの現状から，UAF 攻撃を防止する様々な手法が提案されている [2]–[17]．しかし，ランタイムで UAF 攻撃を防止できる手法は少ない．また，既存の多くの手法は，メモリ使用効率が悪い問題がある．

そこで，本稿では，解放されたメモリ領域の再利用を一定期間禁止するライブラリを実現し，ランタイムで UAF 攻撃を防止する手法を提案する．UAF 攻撃は，オブジェクトのメモリ領域の解放直後に，その解放したメモリ領域を再利用する特徴がある．

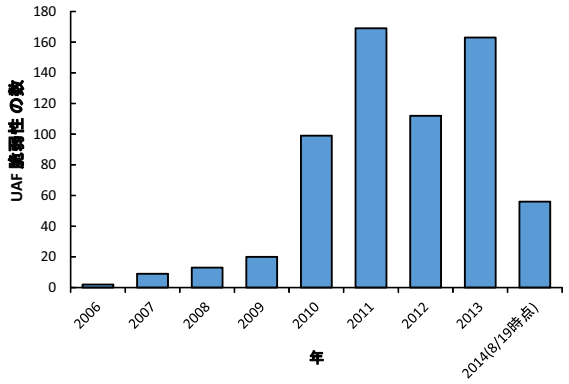

図 1 CVE に登録されている UAF 脆弱性の数

提案ライブラリを適用すると，解放されたメモリ領域の再利用を一定期間禁止するため，上記の UAF 攻撃を防止できる．また，提案したライブラリは，保護対象のプログラムを改変せずに適用できるため，各プログラムへの適用性が高い．さらに，メモリ領域を再利用できるため，メモリ使用効率が高い．

本研究による貢献は，以下の 2 点である．

(1) ライブラリを改変することでランタイムで UAF 攻撃を防止する手法を提案した．

(2) UAF 攻撃の防止において，プログラムを改変することなく，導入の容易な手法を提案した．

2 既存の UAF 攻撃を防止する研究

2.1 UAF 攻撃

図 2 に，Linux/x86_64 での UAF 脆弱性を含むサンプルコードを示し，そのときのメモリレイアウトを図 3 (図 3 は 32 bit で記述) に示す．23 行目で，Addnum クラスのオブジェクトを作成する．Addnum オブジェクトは，24 行目で仮想関数 (print) を利用している．仮想関数は，レジスタ経由で vtable を参照し，実行される．25 行目で Addnum オブジェクトを解放している．その後，29 行目で Addnum オブジェクトと同じサイズの領域の確保を実施すると，メモリの再利用により，解放した Addnum オブジェクトのメモリ領域が図 3 のように確保される．31 行目において，再利用により確保した領域を書き換えることで，Addnum オブジェクトの vtable へのアドレスを格納していた箇所に，シェルコードのポインタを格納するアドレスが書き込まれる．これにより，33 行目のダングリングポインタの参照により，Addnum オブジェクトの仮想関数の実行と同様の手順が実施され，シェルコードが実行される．

```
1  #include <cstdio>
2  #include <cstdlib>
3  #include <cstring>
4  #include <unistd.h>
5  using namespace std;
6
7  class Addnum
8  {
9    int num1, num2, result;
10 public:
11   Addnum(char *arg1, char *arg2) {
12     num1 = atoi(arg1);
13     num2 = atoi(arg2);
14     result = num1 + num2;
15   }
16   virtual void print() {
17     printf("result = %d\n", result);
18   }
19 };
20
21 int main(int argc, char *argv[])
22 {
23   Addnum *addnum = new Addnum(argv[1], argv[2]);
24   addnum->print();
25   delete(addnum);
26
27   char shellcode[] = "\x48\x31\xd2\x52\x48\xb8\
       x2f\x62\x69\x6e\x2f\x2f\x73\x68\x50\x48\
       x89\xe7\x52\x57\x48\x89\xe6\x48\x8d\x42\
       x3b\x0f\x05";
28
29   char *buf = new char[24];
30
31   memcpy(buf,"\x18\x20\x60\x00\x00\x00\x00\x00\
       x40\xe2\xff\xff\xff\x7f\x00\x00",16);
32
33   addnum->print(); // dangling pointer
34
35   delete[] buf;
36   return 0;
37 }
```
図 2 Linux/x86_64 での UAF 脆弱性を含むコード

このように，UAF 攻撃では，オブジェクトのメモリ領域を解放した直後に，解放したメモリ領域を再利用することで，その後に参照されるダングリングポインタにより任意コードを実行できる．

2.2 ダングリングポインタの検知による防止

文献 [2], [3], [4], [5] は，動的バイナリ変換，シャドウメモリ，およびテイント解析などを利用し，事前にダングリングポインタを発見することで，UAF 攻撃を防止する．文献 [6], [7] は，コンパイル時にダングリングポインタを検知するコードを追加することで，実行時に UAF 攻撃を検知する．文献 [6], [8], [9] は，プログラムのバイナリを静的解析することで，ダングリングポインタを検知する．このように，プログラムの運用前に UAF 攻撃の起点となるダングリングポインタの一部を発見することで，UAF 攻撃を防止できる．

2.3 ライブラリの置き換えによる防止

文献 [10], [11], [12] は，メモリ確保をページごとに割当てる malloc ライブラリに置き換えることで，ランタイムで UAF 攻撃を防ぐ．しかし，新しく確保する領域は，ページ単位で確保されるため，メモ

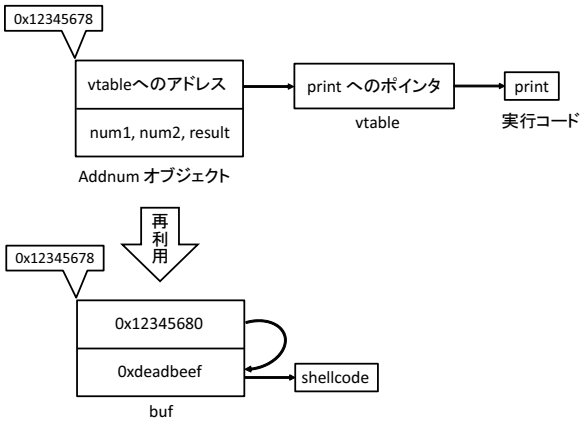

図 3 UAF 攻撃の際のメモリレイアウト

リ使用効率が悪い．

文献 [13] は，同じサイズとアライメントのオブジェクトのみ同じメモリ領域を再利用できる手法 (type-safe) のライブラリを提案している．UAF 攻撃を達成する場合，解放したオブジェクトとは異なるタイプのデータを書き込む．このため，異なる UAF 攻撃を防止できる．しかし，同じサイズとアライメントのオブジェクト以外は再利用できないため，メモリ使用効率が悪い問題がある．

文献 [14], [15] は，メモリ確保をランダムな領域から確保し，連続する領域に確保させない．これにより，UAF 攻撃を防止する．

2.4 Application Program の改変による防止

文献 [16], [17] は，Internet Explorer (以降，IE と略す) の UAF 攻撃の対策であり，Application Program (以降，AP と略す) から新たに対策済みの関数を呼び出すことで，UAF 攻撃を防止する．文献 [16] は，主要な関数が作成するオブジェクトを IE 固有のヒープ領域 (プロセスヒープ) に作成せず，新たに確保したプライベートヒープに作成する．これにより，ヒープ領域が独立されるため，UAF 攻撃を困難にできる．文献 [17] は，解放したメモリ領域を一定期間解放しないことで，UAF 攻撃を困難にする．この期間は，解放したメモリ領域の合計サイズが閾値以上 (100 KB 以上) になるまでである．解放するメモリの合計サイズが閾値以上になった場合，解放を待機していたメモリ領域をすべて解放し，再利用可能にする．しかし，この手法を IE 以外のプログラムに適用するには，各プログラムの改変が必要であり，工数が増加する問題がある．

2.5 既存研究の問題点

既存の研究の問題点を以下に示す．

(問題点 1) ランタイムで UAF を防止できない
2.2 節の研究は，プログラムの運用前に UAF 攻撃の起点となるダングリングポインタを検知することで，UAF 攻撃を防ぐ．このため，運用前に発見できなかったダングリングポインタを悪用された場合，UAF 攻撃を防止できない問題がある．

(問題点 2) メモリ使用効率が悪い
2.3 節の研究は，セキュアな malloc ライブラリに置き換えるため，UAF 攻撃をランタイムに検知できる．しかし，メモリ使用効率化の問題がある．

(問題点 3) プログラムのコードを改変する必要あり
2.4 節の研究は，IE のプログラムを改変し，新たに追加した関数を呼び出すことで，UAF 攻撃を防ぐ．このため，プログラムを改変する必要がある．

本稿では，これらの 3 つの問題を解決する UAF 攻撃防止手法を提案する．

3 ライブラリの改変により UAF 攻撃を防止する手法

3.1 考え方

事前調査として，CVE-2012-4792，CVE-2012-4969，CVE-2013-3893，および CVE-2014-1776 の脆弱性を利用した攻撃を調査した．調査結果より，UAF 攻撃におけるメモリ再利用のタイミングは，オブジェクトの解放直後に再利用することが分かった．解放直後に再利用する理由は，別の処理により，解放したオブジェクトの領域を再利用させないためである．これより，UAF 攻撃の多くは，オブジェクトの解放直後にメモリを再利用する．

そこで，解放したメモリ領域を再利用させないことで，UAF 攻撃を防げることに着目する．しかし，メモリ領域を永久に再利用させない場合，メモリ使用効率が悪くなる．また，新たに領域を確保する処理が増加し，オーバヘッドが大きくなる問題がある．これに対処するため，解放した領域の再利用を禁止を一定期間の間のみとする．

3.2 基本方式

UAF 攻撃は，オブジェクトの解放直後にメモリを再利用するため，提案手法は，既存のライブラリを改変し，解放したメモリ領域の再利用を一定期間禁止することで UAF 攻撃を防止する．

再利用を可能にする条件を以下に示す．

(1) 解放したメモリ領域の合計サイズが指定したサイズ以上である

(2) 解放したメモリ領域が前の領域と結合している

条件 (1) を満たしたとき，条件 (2) を満たすメモリ領域を最大で指定した合計サイズの半分まで解放する．条件 (1) は，文献 [17] と同様の手法であり，メモリの解放直後にその解放したメモリ領域を再利用されることを防げる．しかし，文献 [17] は，閾値の設定が 100 KB で固定されているため，閾値で設定したサイズ分のメモリを確保され，解放された場合，UAF 攻撃が達成される可能性がある．そこで，提案手法では，閾値に設定する合計サイズを大きくすることでエントロピーが増加し，攻撃を困難にできる．また，閾値を指定する範囲内でランダマイズ化することで，より UAF 攻撃を困難にできる．さらに，再利用可能時は，解放したメモリ領域の半分のみを再利用可能にすることで，一部のメモリ領域の再利用を遅らせ，より UAF 攻撃の達成を困難にできる．

続いて，(2) の条件を追加することで，ダングリングポインタを参照するメモリ領域までのオフセットを算出しなければ UAF 攻撃は成功しないため，UAF 攻撃を困難にできる．

ブラウザなどの大規模なプログラムは，C++ のプログラムで実現されていることが多い．C++ のプログラムは，Linux では，libstdc++ ライブラリをリンクし，実行する．libstdc++ ライブラリは，内部で glibc をインクルードしているため，new 演算子や delete 演算子は，最終的に malloc や free を呼ぶ．このため，glibc を改変するのみで，libstdc++ ライブラリにも適用でき，C++ プログラムにも提案手法を導入できる．

既存のライブラリを改変する利点として，多くの OS に導入されているため適用しやすいこと，ライブラリの置き換えだけで既存プログラムを改変せずに適用できること，および工数を小さくできることが挙げられる．また，提案手法は，既存のライブラリの malloc アルゴリズムのみを改変するため，他のライブラリ関数に影響しない．本研究では，多くの Linux ディストリビューションで実用されている glibc を対象とした実現方法について述べる．

3.3 利用形態

提案手法は，共有ライブラリで実現するため，提案手法のライブラリの導入は，既存のライブラリと提案手法のライブラリを置き換えるか，AP のリンク先のライブラリを提案手法のライブラリに指定すればよい．

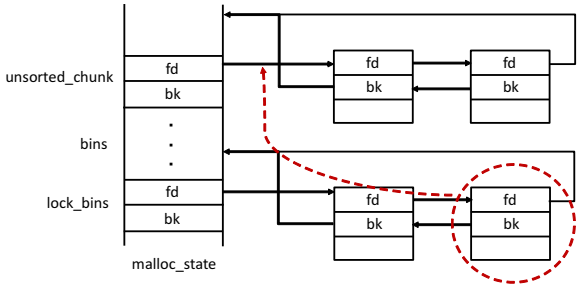

図 4 提案手法のメモリ解放時のデータ構造の処理

3.4 提案手法の詳細

提案手法は，malloc アルゴリズムのメモリ解放時の処理 (free 時) のみを改変する．既存の malloc アルゴリズムのメモリ解放時 (mmap() で確保していない領域の処理) の処理を以下で説明する．

(1) 解放するメモリ領域 (chunk) のサイズを取得し，fastbins または unsorted_bins に chunk を登録

glibc は，解放したメモリ領域を chunk と呼ばれるデータ構造として malloc ライブラリ内の malloc_state 構造体で管理される．解放した chunk は，fastbins や unsorted_bins というリストに登録され，管理される．解放する chunk のサイズが 128 bytes 以下の場合，fastbins に登録される．128 bytes より大きい場合，前後のメモリ領域が未使用なら結合し，unsorted_chunk に登録される．メモリ確保の際，これらのリストを確認し，適切な chunk を確保する．

続いて，提案手法のメモリ解放時のデータ構造の処理を図 4 に示し，以下で説明する．

(1) 解放するメモリ領域 (chunk) を提案手法が用意したリスト (lock_bins) の先頭に格納

(2) lock_bins に格納された chunk の合計サイズが閾値以上になった場合，lock_bins の後尾からメモリ上の前の chunk と結合している chunk を最大で指定した合計サイズの半分まで unsorted_chunk に移動

lock_bins は，malloc_state 構造体に新たに追加した．また，chunk は，自身のサイズを管理する変数を管理しているため，chunk の合計サイズを算出できる．再利用を可能にする合計サイズの閾値は，1 MB 以上を設定することで，UAF 攻撃を困難にで

きると考える．Linux/x86_64 に適用した glibc の場合，128 KB 以上のメモリ確保は，mmap() で確保される．このため，閾値を 1 MB 以上とすることで，7 個以上の chunk が lock_bins に登録される．これより，攻撃対象のオブジェクトの解放直後にその領域を確保できないため，UAF 攻撃を困難にできる．さらに，chunk を再利用可能にするたびに，閾値で設定する合計サイズを特定範囲内でランダマイズ化することで，より UAF 攻撃を困難にできる．

解放する chunk は，lock_bins に格納される際，前の chunk が未使用の場合，それらと結合する．前の chunk と結合した chunk は，再利用可能な chunk となる．lock_bins から解放された chunk は，unsorted_chunk に登録される．unsorted_chunk は，既存の malloc ライブラリに存在し，メモリ領域を確保する際，unsorted_chunk から適応する chunk がないか調査する．適応しない chunk は，chunk サイズごとにリスト管理している bins に格納される．これにより，chunk は再利用可能となる．

3.5 期待される効果

提案手法の導入により期待される効果を示す．

(1) ランタイムで UAF を防止可能

提案手法は，ライブラリで実現しているため，ランタイムで UAF 攻撃を検知できる．このため，UAF 脆弱性のバグが存在するプログラムを動作させても，提案手法の導入により，UAF を防げる．また，3.2 節の方式により，既存のランタイムで UAF 攻撃を防止する手法より UAF 攻撃を困難にできる．

(2) メモリ使用量が小さい

提案手法は，解放されたメモリ領域の確保を一定期間禁止する．このため，一定期間を過ぎれば，再利用できる．これにより，メモリ使用量の増加を抑えることができる．

(3) プログラムのコードを改変する必要なし

提案手法は，共有ライブラリで実現しているため，保護対象のプログラムのコードを改変する必要はない．このため，プログラムの開発者は，UAF 脆弱性のバグを気にせず公開できる．

(4) 適用性が高い

提案手法は，ソースコードが公開されているライブラリであれば，ライブラリを改変することで提案手法を実現できる．また，Windows においても DLL インジェクションのような手法により，提案手法を適用できると考える．

4 実装

提案手法は，glibc-2.13 の malloc アルゴリズムを改変することで実現した．また，今回の実装では，解放するメモリ領域が前のメモリ領域と結合したか否かの判定は実装していない．このため，前の chunk と結合していない chunk も unsorted_chunk に登録される．解放するメモリ領域が前のメモリ領域と結合したか否かの判定は，今後の課題とする．

5 評価

5.1 評価環境

5.2 節と 5.3 節の Web サーバの評価環境は，CPU は Intel Core i7-3770 (3.40 GHz)，メモリは 4 GB，OS は Linux 3.4.9/x86_64，ディストリビューションは Debian 7.0 である．

5.3 節の malloc-test と 5.4 節の評価環境は，CPU は Intel Core i5 M 430 (2.27 GHz)，他は，上記と同様である．また，提案手法で改変した glibc のバージョンは，glibc-2.13 である．

5.2 UAF 攻撃の防止実験

提案手法を適用し，図 2 のプログラムを用いて，UAF 攻撃を防止できることを明らかにする．ASLR と DEP を無効にした状態で実行すると，27 行目のシェルコードが実行される．図 5 に，提案手法を適用する前後の状態で，図 2 のプログラムを実行した結果を示す．図 5 -(a) は，Addnum オブジェクトと buf が同じメモリ領域に配置され，ダングリングポインタの参照により UAF 攻撃が成功し，図 2 の 27 行目のシェルコードが実行されている．一方，図 5 -(b) Addnum オブジェクトと buf は異なるメモリ領域に配置され，UAF 攻撃を防止できている．ただし，ダングリングポインタが参照した箇所は，アクセス不可領域となっているため，Segmentation fault で終了している．

以上より，提案手法を導入により UAF 攻撃を防止できたことを明らかにした．しかし，ダングリングポインタの参照により，メモリ内容が漏洩してしまう危険性がある．そこで，メモリ解放する際に，解放した領域をゼロフィルするか，アクセス不能属性を設定することで，メモリ内容を漏洩する攻撃に対策できる．また，デバッグ割り込みを発生させる 0xcc で埋めることで，ダングリングポインタを発見できる．このように，バグの検知としても提案手法を活用できる．

```
yuta@debian:~$ ./uaf 100 10
result = 110
Addnum = 0x602010
buf = 0x602010
$
```

(a) 提案ライブラリ適用前

```
yuta@debian:~$ LD_PRELOAD="/usr/local/test2
/lib/libc.so.6" ./uaf 100 10
result = 110
Addnum = 0x602010
buf = 0x602030
Segmentation fault
```

(b) 提案ライブラリ適用後

図 5 UAF 攻撃の防止実験

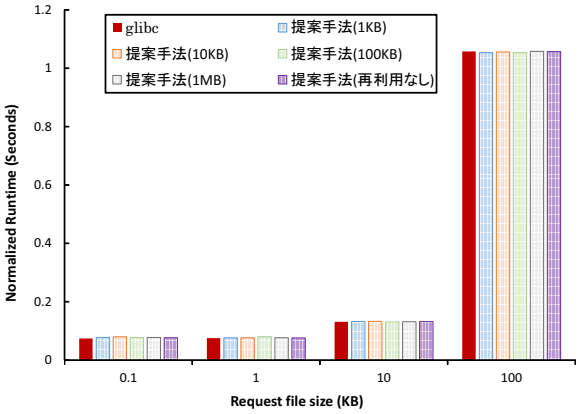

図 6 thttpd の処理時間

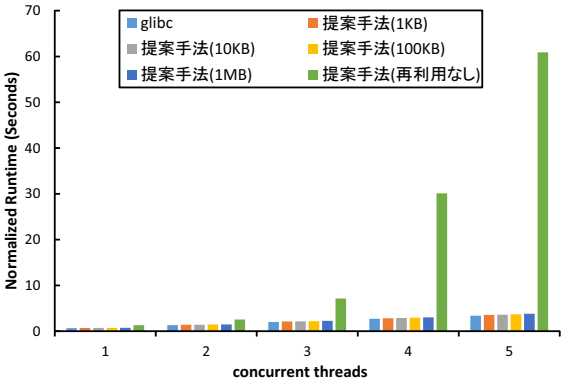

図 7 100 bytes のメモリ確保・解放の処理時間

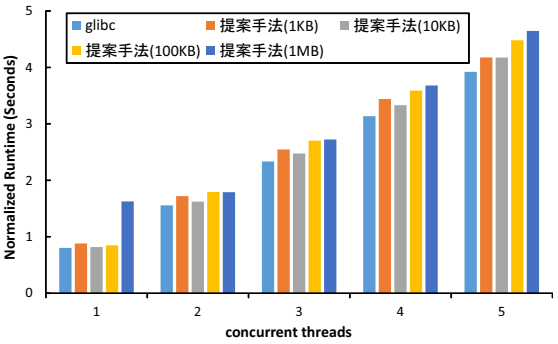

図 8 512 bytes のメモリ確保・解放の処理時間

5.3 提案手法の導入によるオーバヘッドの測定

提案手法のオーバヘッドを評価するため，glibc，提案手法 (閾値 1 KB)，提案手法 (閾値 10 KB)，提案手法 (閾値 100 KB)，提案手法 (閾値 1 MB)，および提案手法 (再利用なし) の導入において，Web サーバの処理時間を測定し，比較した．再利用なしの提案手法は，解放したメモリを unsorted_chunk へ登録しないことで，実現した．評価では，Web サーバとして thttpd 2.25b を利用し，ベンチマークは ApacheBench を利用した．評価内容は，thttpd に同時に 50 のアクセスを行い，100 bytes，1 KB，10KB，100 KB，1 MB のファイルに対し，それぞれ 1,000 回リクエストを要求する際の処理時間を測定した．評価結果を図 6 に示す．図 6 に示す通り，提案手法のいずれの閾値においても，glibc との処理時間はほぼ同じである．これより，提案手法の AP への性能の低下は小さいことが分かる．

また，malloc-test ベンチマーク [18] を利用し，マルチスレッドでのメモリ確保・解放を繰り返す際の処理時間を測定した．ここでは，100 bytes，512 bytes，1,024 bytes のそれぞれのメモリサイズを確保・解放する処理を 1,000 万回繰り返した際の処理時間を測定した．また，スレッドは，1 から 5 つまで増やした．評価結果を図 7，8，9 に示す．100 bytes のメモリ確保・解放において，再利用しない提案手法の処理時間は，再利用する提案手法と比べ，処理時間が大きいことが分かる．これは，メモリ確保の際，毎回新しい領域を取得するためである．再利用しない提案手法は，512 bytes での評価から，物理メモリの消費量が増加し，測定できない状態となった．

再利用する提案手法は，再利用の閾値が大きいほど，処理時間が大きいことが分かる．また，改変前の glibc と提案手法との実行時間の差異は，小さいことが分かる．

以上の評価結果より，提案手法の導入によるオーバヘッドは小さいことを明らかにした．

5.4 提案手法の導入によるメモリ使用量の測定

提案手法と既存の malloc アルゴリズムのメモリ使用量を測定し，比較する．malloc-test を利用し，512 bytes のメモリサイズ を 5 スレッドで確保・解放する処理を 1,000 万回繰り返す際のメモリ使用量を測定した．評価結果を図 10 に示す．図 10 の通り，提案手法 (1 MB) 以外のメモリ使用量は，ほぼ

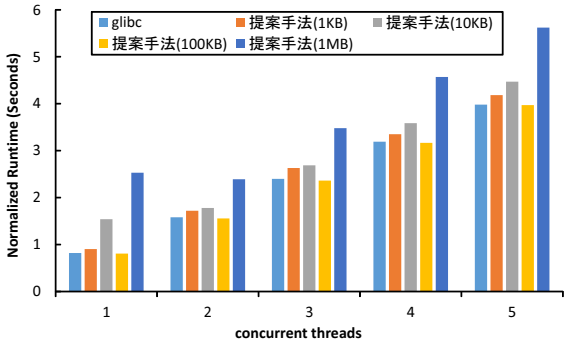

図 9　1,024 bytes のメモリ確保・解放の処理時間

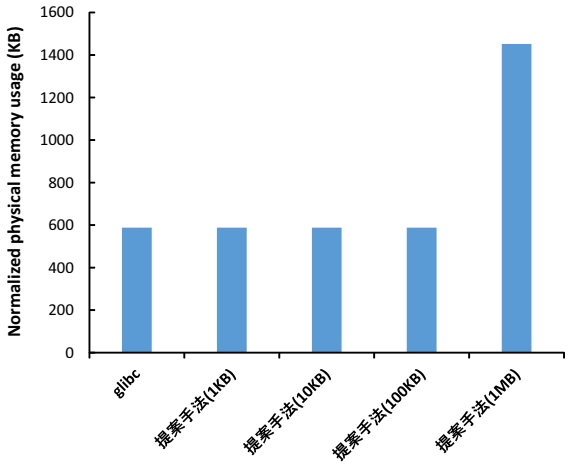

図 10　malloc-test のメモリ使用量

同じである．提案手法 (1 MB) のメモリ使用量が大きくなった原因は，1 MB までメモリを再利用できなかったためである．

また，firefox 31.0 を使用し，Selenium IDE を使用し，10 個の Web サイトを連続で閲覧した際のメモリ使用量を測定した．評価結果を図 11 に示す．glibc も 提案手法も 280 ～ 320 MB 内のメモリ使用量であり，ほとんど差異はない．

以上の評価結果より，提案手法の導入によるメモリ使用量は，小さいことを明らかにした．

5.5　既存手法との比較

2.3 節と 2.4 節の一部の手法と提案手法との比較を表 1 に示す．DieHarder [15] は，ソースコードが公開されているため，DieHarder の処理時間とメモリ使用量を測定した．測定は，malloc-test を利用し，512 bytes のメモリサイズ を 5 スレッドで確保・解放する処理を 1,000 万回繰り返す際のメモリ使用量を測定した．DieHarder の処理時間は，11.67 s であり，提案手法 (1 MB) の約 2.5 倍である．DieHarder のメモリ使用量は，3,608 KB であり，提案手法 (1 MB) の約 2.5 倍である．

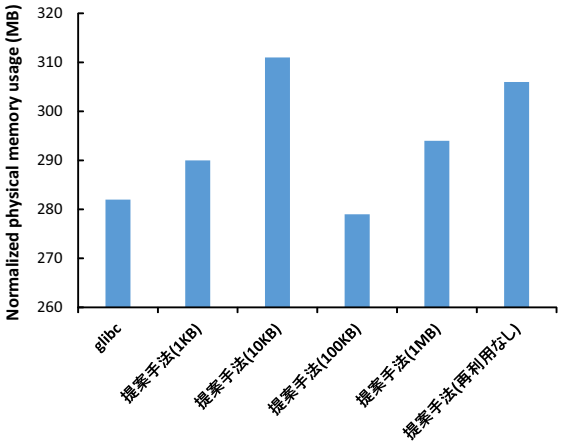

図 11　firefox でのメモリ使用量

6　考察

6.1　提案手法の異なる OS への適用

提案手法を実現した glibc は，UNIX 系の OS では，すべて適用できる．また，glibc 以外のライブラリにおいても，メモリの解放処理を改変することで，提案手法を実現できると考える．Windows においても，文献 [15] の手法のように，DLL として提案手法を実現することで，提案手法を適用できる．

6.2　提案手法を回避する攻撃方法

提案手法は，一定期間で解放した領域を再利用する．このため，確保と解放を繰り返すことで，設定した閾値を推測し，提案手法を回避して攻撃される可能性がある．しかし，上記の攻撃が達成できる条件を満たすことは，困難である．

1 つ目に，攻撃者は，解放したメモリ領域の保持数と合計サイズを把握する必要がある．また，合計サイズは，再利用可能にする度に，指定した範囲内でランダムな値が設定されるため，サイズの予測は困難である．

2 つ目に，解放するメモリ領域は，前のメモリ領域と結合しているため，ダングリングポインタの参照箇所までのオフセットを算出する必要がある．前のメモリ領域も結合している可能性があるため，攻撃者は，これらの結合を推測し，適切なサイズのメモリ領域を確保する必要がある．

7　おわりに

解放されたメモリ領域の再利用を一定期間禁止するライブラリを実現し，ランタイムで UAF 攻撃を

表 1 既存手法と提案手法との比較

	DUMA [12]	DieHarder [15]	DelayFree [17]	Cling [13]	提案手法
メモリ再利用可能か	不可	可能	可能	可能	可能
再利用の周期	なし	ランダム	解放したメモリのサイズ	type-safe	(1) 解放したメモリのサイズ (2) 前のメモリ領域と結合判定
プログラムの改変	なし	なし	あり	なし	なし
対象 OS	Linux	Linux, Windows	Windows	Linux	Linux

防止する手法を提案した．提案したライブラリは，保護対象のプログラムを改変せずに適用でき，解放されたメモリ領域の再利用を一定期間禁止することで，UAF 攻撃を防止できる．ライブラリで UAF 攻撃を防止できるため，保護対象のプログラムを改変せずに適用できる．

評価では，提案ライブラリを導入することで，UAF 攻撃を防げたことを明らかにした．また，既存の malloc ライブラリとの性能の比較では，メモリ使用量が小さいことを明らかにした．

今後の課題として，さらなる評価，メモリ使用効率化，および Windows ライブラリへの適用がある．

参考文献

[1] Common vulnerabilities and exposures, https://cve.mitre.org/index.html.

[2] 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).

[3] 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).

[4] 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).

[5] 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).

[6] 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).

[7] Eigler, C.F.: Mudflap: Pointer Use Checking for C/C++, http://gcc.fyxm.net/summit/2003/mudflap.pdf.

[8] 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).

[9] Dewey, D. and Giffin, T.J.: Static Detection of C++ Vtable Escape Vulnerabilities in Binary Code, Proc. 19th Network and Distributed System Security Symposium, pp.1–14 (2012).

[10] Pageheap, http://technet.microsoft.com/ja-jp/library/cc835607.aspx.

[11] Electric fence, http://elinux.org/Electric_Fence.

[12] DUMA, http://duma.sourceforge.net/.

[13] Akritidis, P.: Cling: A Memory Allocatorto Mitigate Dangling Pointers, Proc. 19th USENIX Conference on Security, pp.177–192 (2010).

[14] Lvin, V.B., Novark, G., Berger, E.D. and Zorn, B.G.: Archipelago: Trading Address Space for Reliability and Security, Proc. 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.115–124 (2008).

[15] Novark, G., Berger, E.D.: Dieharder: Securing the heap, Proc. 17th ACM Conference on Computer and Communications Security, pp.573–584, (2010).

[16] Tang, J.: Isolated heap for inter-net explorer helps mitigate uaf exploits, http://blog.trendmicro.com/trendlabs-security-intelligence/isolated-heap-for-internet-explorer-helps\-mitigate-uaf-exploits/.

[17] Tang, J.: Mitigating uaf exploits with delay free for internet explorer, http://blog.trendmicro.com/trendlabs-security-intelligence/mitigating-uaf-exploits-with-delay\-free-for-internet-explorer/.

[18] Lever, C. and Boreham, D.: Malloc() Performance in a Multithreaded Linux Environment, Proc. 9th Annual Conference on USENIX Annual Technical Conference, pp.301–311 (2000).