Vol.2015-CSEC-69 No.21
Vol.2015-IOT-29 No.21
2015/5/22

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

山内 利宏[1]　池上 祐太[1]

概要：脆弱性を悪用した攻撃への対策として，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（UAF 攻撃）の防止手法を文献 [1] で提案した．提案手法は，ライブラリを改変することで実現し，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により，UAF 攻撃を防止する．本稿では，特に Windows での提案手法の実現方式について述べる．また，Windows において実際にサイバー攻撃に使用された UAF 攻撃で評価した結果，および Linux と Windows でオーバヘッドを評価した結果を報告する．

1. はじめに

近年，Use-After-Free 脆弱性攻撃 (以降，UAF 攻撃と略す) が増加している．UAF 攻撃とは，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攻撃である．文献 [2] 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．図 1 から，2010 年以降，UAF 脆弱性の発見数は急増している．Microsoft 製品への脆弱性攻撃においても，UAF 攻撃の利用が最も高い [3]．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．

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 [4]–[20]．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，文献 [1] 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ことで，ランタイムで UAF 攻撃を防止する手法を提案した．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．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，再利用可能とする期間の推測をより難しくし，UAF 攻撃を高い確率で防止できる．また，提案手法は，ライブラリで実現されており，保護対象のプログラムを改変せずに適用できるため，プログラムへの適用は比較的容易である．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．しかし，文献 [1] では，Linux での基本的な実現方式と Linux での一部の評価しか述べられていない．

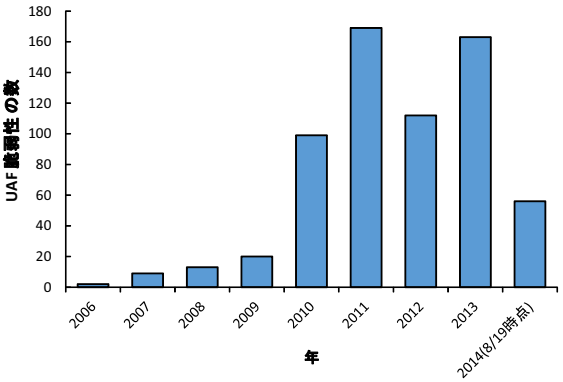

図 1　CVE に登録されている UAF 脆弱性の数

本稿では，Linux における詳細な実現方式，および Windows における提案手法の実現方式について述べる．また，Linux でのさらなるオーバヘッドの評価結果，Windows での実際にサイバー攻撃に使用された UAF 攻撃での評価結果とオーバヘッドの評価結果を報告する．

2. 既存研究

文献 [1] に述べている 3 つの問題点について簡単に述べ，新たに追加した一つの問題点について述べる

[1] 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University 3-1-1 Tsushima-naka, Okayama, 700-8530 Japan

2.1 ダングリングポインタの検知による防止

プログラムの実行前に 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研究として，以下の手法がある．文献 [4], [5], [6], [7], [8] 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研究である．文献 [9], [10] 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 [9], [11], [12] 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．

2.2 ライブラリの置き換えによる防止

ライブラリのメモリを確保処理を変更することで，UAF 攻撃を防ぐ手法として，以下のものがある．文献 [13], [14], [15] 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，メモリ使用効率が悪い．文献 [16] 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 [17], [18] 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．

2.3 アプリケーションプログラムの改変による防止

アプリケーションプログラム（以降，AP と略す）の改変により，UAF 攻撃を防ぐ手法について述べる．文献 [19], [20] は，Internet Explorer (以降，IE と略す) の UAF 攻撃の対策であり，AP 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 [19] 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 [20] 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．

2.4 vtable の改竄を防止する手法

文献 [21], [22], [23] は，vtable の改竄を防止する手法により，UAF 攻撃を防止する．UAF 攻撃の多くは，vtable に格納されているポインタを書き換え，任意コードを実行する．しかし，この手法では，vtable を改竄しない UAF 攻撃には対処できない問題がある．また，事前に保護対象プログラムのバイナリを書き換える必要があり，バイナリの形式に依存するため，適用には工数がかかる．

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
2.1 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
2.2 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
2.3 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．

(問題点 4) 適用範囲が限定的
2.4 節の研究は，vtable のみに着目しているため，vtable を生成しない言語で作成されたプログラムを対象とした UAF 攻撃は，防げない問題がある．

本稿では，これらの 4 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．

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．

Vol.2015-CSEC-69 No.21
Vol.2015-IOT-29 No.21
2015/5/22

3.2 Linux での実現方式

3.2 節では，文献 [1] で述べた Linux の実現方式について簡単に述べる．また，3.3 節では，文献 [1] では実現できていなかった前後の領域と結合した領域を再利用する方式について詳しく述べる．

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後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 [20] 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 [20] 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，攻撃をより困難にする．また，設定する合計サイズの閾値を，指定する範囲内でランダムに設定することで，より UAF 攻撃を困難にする．さらに，再利用可能なメモリを選択する際に，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する．

これに加え，(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適切に算出しなければ UAF 攻撃は成功しないようにし，UAF 攻撃をさらに困難にしている．

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．ここでは，多くの Linux ディストリビューションで実用されている glibc (x86_64) を対象とした実現方法について述べる．

3.3 提案手法の詳細 (Linux)

提案手法は，malloc アルゴリズムのメモリ解放時の処理（free 時）のみを改変することで実現できる．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．

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 構造体で管理する．解放した chunk は，fastbins や unsorted_chunks 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 byte 以下の場合，fastbins に登録される．128byte 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_chunks に登録される．

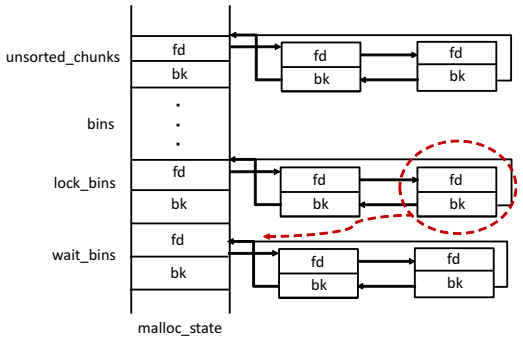

図 2 提案手法のメモリ解放時のデータ構造の処理

メモリ確保の処理では，これらのリストを確認し，適切な chunk を確保する．

次に，提案手法のメモリ解放時の処理を**図 2** に示し，以下で説明する．

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

(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後の chunk と結合，または前後どちらかの 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解放し，unsorted_chunks に登録する．

(3) (2) で前後の chunk，または前後どちらかの chunk と結合できなかった場合，その chunk に結合可能属性を付与し，提案手法で用意したリスト (wait_bins) に格納する．

lock_bins と wait_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．lock_bins から chunk を取り出す際，unsorted_chunks と wait_bins から結合できる chunk を探す．このとき，結合できる chunk が存在しない場合，lock_bins から取り出した chunk に結合可能属性を付与し，wait_bins に格納する．lock_bins と

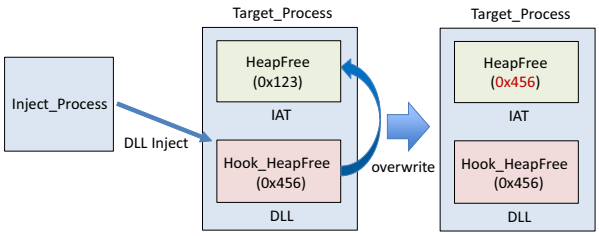

図 3　Windows での提案手法の実現方式

wait_bins から解放された chunk は，unsorted_chunks に登録される．malloc ライブラリは，メモリ領域を確保する際，まず，unsorted_chunks から適応サイズの chunk がないか調査する．適応しなかった 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．

3.4　利用方法

提案手法は，ライブラリに適用する．このため，提案手法を適用したライブラリの導入は，既存のライブラリと提案手法を適用した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を適用したライブラリに指定すればよい．

3.5　Windows での実現方式

Windows での再利用可能条件は，3.2 節と同様である．Windows では，主要なメモリ管理を kernel32.dll，ntdll.dll，ntoskrnl.exe で実現している．kernel32.dll と ntdll.dll は，ユーザレベルでのメモリ操作を処理し，ntoskrnl.exe は，カーネルレベルでのメモリ操作を処理している．Linux の glibc と同様の処理を実施しているのは，kernel32.dll と ntdll.dll である．また，ヒープ領域を解放する処理は，kernel32.dll で定義されている HeapFree() を使用することが多い．そこで，HeapFree() をフックすることで，一定期間メモリ再利用を禁止させる．

3.6　提案手法の詳細 (Windows)

プロトタイプを DLL (Dynamic Link Library) インジェクションと Windows API フックにより，実現した．**図 3** に，Windows での提案手法の実現方式を示す．DLL インジェクションとは，他プロセスに DLL をマッピングし，マッピングした DLL の処理を他プロセス内で実行させる手法である．Windows API フックとは，Windows API の呼び出しをフックし，任意の処理を実行させる手法である．Windows API フックでは，IAT (Import Address Table) フックを使用した．IAT には，プロセスのロード時に，DLL からエクスポートされる API の関数のアドレスが格納される．IAT フックは，IAT に格納された API の関数をアドレスを任意の関数のアドレスに書き換えることで，API をフックする手法である．

提案手法は，まず，DLL インジェクションにより，IAT フックを実施する DLL (Hook.dll) をターゲットプロセスにマッピングする．Hook.dll は，IAT に格納されている HeapFree 関数 のアドレスを自身の関数のアドレスに書き換える．書き換えた先の関数では，HeapFree 関数 の引数を取得し，リングバッファに引数を格納する．その後，解放したメモリ領域の合計値が閾値以上になった場合，リングバッファから引数を取り出し，本来の HeapFree 関数を呼び出し，解放したメモリの半分を再利用可能にする．

これらの処理により，Linux と同等の手法を Windows で実現した．ただし，前後のメモリ領域と結合したかの判定は未実装であるため，今後の課題とする．また，解放したメモリ領域のサイズの取得は，処理が煩雑になるため，解放したメモリ領域の個数を閾値として利用した．個数として利用する場合でも，合計サイズで閾値を設定する場合と同様に，個数を大きくすることやランダマイズ化することで，エントロピーが増大し，攻撃を困難にできる．

4.　評価

4.1　評価環境

評価環境は，CPU は Intel Core i7-3770 (3.40 GHz)，メモリは 4 GB，OS は Linux 3.13.0-45-generic/x86_64 (Ubuntu 14.04 LTS)，Windows 7 (64 bit) である．また，改変した glibc のバージョンは，glibc-2.19 である．

4.2　Linux でのオーバヘッドの測定

ここでは，glibc と提案手法を比較するため，数種類のベンチマークで性能を測定した．1 つ目に，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 100 KB と 1 MB) の導入において，ブラウザベンチマークを用いて測定した．ここでは，ブラウザとして Firefox と Chrome を用い，Google's Octane 2.0[24]，Apple's SunSpider 1.0.2[25]，Mozilla's Kraken 1.1[26]，Microsoft's LiteBrite[27]，FutureMark's Peacekeeper[28]，Mozilla's Dromaeo[29] の 6 種類のブラウザベンチマークを測定した．測定結果は，ベンチマークを 3 回実行した際の平均スコアである．**図 4** と **図 5** に，Firefox と Chrome での glibc に対する性能オーバヘッドを示す．

図 4 より，Firefox の場合では，100KB と 1MB の双方において，オーバヘッドは，1.8%未満である．100KB と 1MB を比較すると，再利用までの閾値が大きい 1MB の方がオーバヘッドが大きい．これは，新しく領域を確保する処理が増加したためだと推察できる．また，図 5 より，Chrome の場合では，100KB と 1MB の双方において，オーバヘッドは，3% 未満である．Firefox の場合と同様に，1MB の方がオーバヘッドが大きい．

2 つ目に，malloc-test ベンチマーク [33] 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

表 1　Overheads of malloc-test.

File size	lib	thread num				
		1	2	3	4	5
100B	glibc	0.335	0.674	1.02	1.36	1.71
	100KB	0.398 (18.8%)	0.802 (19%)	1.2 (17.6%)	1.612 (18.4%)	2.015 (18.1%)
	1MB	0.399 (19.1%)	0.806 (19.6%)	1.205 (18.1%)	1.613 (18.5%)	2.02 (18.4%)
512B	glibc	0.371	0.748	1.132	1.51	1.885
	100KB	0.425 (14.5%)	0.867 (15.9%)	1.31 (15.7%)	1.754 (16.2%)	2.195 (16.4%)
	1MB	0.437 (17.8%)	0.88 (17.6%)	1.324 (17.1%)	1.765 (16.2%)	2.21 (17.2%)
1024B	glibc	0.374	0.754	1.137	1.518	1.903
	100KB	0.526 (40.6%)	0.998 (32.4%)	1.495 (31.5%)	1.988 (31%)	2.481 (30.4%)
	1MB	0.543 (45.2%)	1.01 (40%)	1.503 (36.6%)	2.01 (32.4%)	2.509 (31.8%)

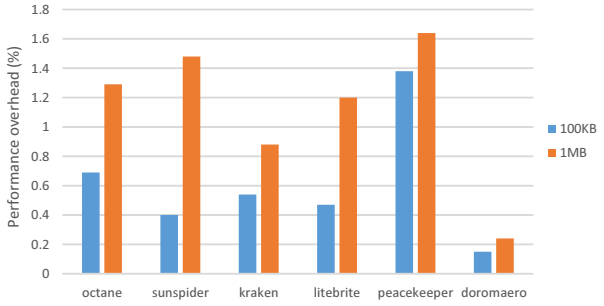

図 4　Overheads of Firefox browser benchmark

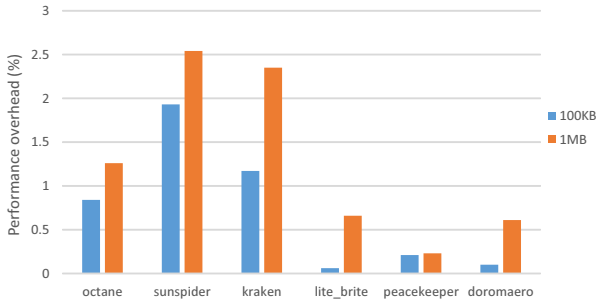

図 5　Overheads of Chrome browser benchmark

間を測定した．ここでは，100 byte，512 byte，1,024 byte 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．提案手法の閾値は，100KB と 1MB の 2 種類である．評価結果を**表 1** に示す．表 1 より，100 byte と 512 byte のメモリ確保・解放において，オーバヘッドは 20% 未満であった．1024 byte のメモリ確保・解放では，オーバヘッドが 約 30%〜45% に増加している．これは，カーネルに新しい領域を要求するシステムコールが増加したことが原因だと推察できる．また，どのサイズにおいても，提案手法の閾値が大きい方がオーバヘッドが大きいことが分かる．

3 つ目に，UnixBench [30], sysbench [31], himeno benchmark [32] を用い，オーバヘッドを測定した．提案手法の閾値は，100KB と 1MB の 2 種類である．評価結果を**表 2** に示す．提案手法は，どのベンチマークにおいても，オーバヘッドは，1%未満である．

表 2　Overheads of UnixBench, sysbench and himeno.

lib	UnixBench (Kbyte/s)	sysbench (s)	himeno (MFLOPS)
glibc	4,139.18	25.98	2,690.24
100KB	4,131.38 (0.19%)	26.21 (0.23%)	2,689.64 (0.02%)
1MB	4,130.57 (0.21%)	26.22 (0.24%)	2,688.05 (0.08%)

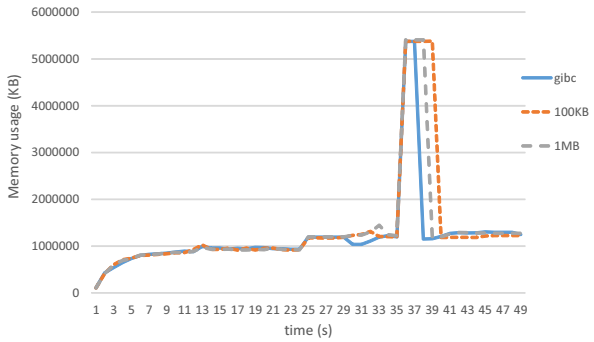

図 6　Overheads of Firefox browser memory usage

以上の評価結果より，提案手法のオーバヘッドは小さいことを明らかにした．また，メモリ確保・解放を大量に繰り返す処理では，性能へ影響が見られることを明らかにした．

4.3　Linux でのメモリ使用量の測定

ここでは，glibc と提案手法を比較するため，ブラウザベンチマークを実行した際のブラウザプロセスの仮想メモリ使用量の推移を測定した．提案手法の閾値は，100KB と 1MB の 2 種類である．**図 6** に，Firefox で Octane を実行した際のメモリ使用量の推移を示す．図 6 より，glibc とほぼ同じメモリ使用量ということが分かる．具体的に，メモリ使用量の最大値は，glibc では，5,375,276 KB，閾値が 100 KB の場合は，5,382,988 KB，閾値が 1MB の場合は，5,407,820 KB であった．Kraken と SunSpider のメモリ使用量の推移を測定したところ，Octane と同様の結果となった．これより，提案手法は，メモリ使用量においてのオーバヘッドも小さいといえる．

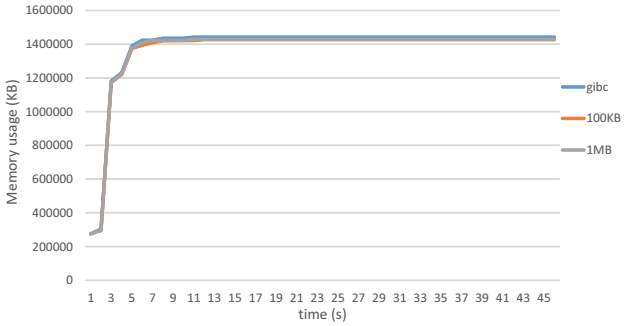

図 7　Overheads of Chrome browser memory usage

また，**図 7** に，Chrome で Octane を測定した際のメモリ使用量の推移を示している．Chrome は，起動時に，複数プロセス (ここでは，6 プロセス) を作成するため，最もメモリ領域を使用しているプロセスのメモリ使用量の推移を抽出した．図 7 より，Chrome の場合においても，提案手法は，glibc とほぼ同じメモリ使用量ということが分かる．具体的に，メモリ使用量の最大値は，glibc では，5,375,276 KB，閾値が 100 KB の場合は，5,382,988 KB，閾値が 1MB の場合は，5,407,820 KB であった．

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．

4.4　Windows での UAF 攻撃の防止実験

Metasploit [34] で配布されている攻撃コードを用いて UAF 攻撃を防止できるか実験した．実験に用いた環境と攻撃コードは，Windows XP の IE7 で CVE-2011-1260 と CVE-2012-4969，Windows 7 の IE10 で CVE-2014-0322 を使用した．提案手法の閾値を設定するため，Linux での提案手法において，閾値を 1MB とした際に，ブラウザベンチマーク実行時のメモリ再利用時にどの程度解放されたメモリ領域を保有しているか調査した．調査結果より，解放されたメモリ領域を 約 3,000 個保有することが分かった．これより，Windows での提案手法の閾値に使用する解放したメモリ領域の個数は，3,000 個とした．

提案手法を IE に適用し，攻撃コードを実行したところ，すべての攻撃コードを防げたことを確認した．

4.5　Windows でのオーバヘッドの測定

提案手法の導入前と導入後でのオーバヘッドを測定する．ここでは，IE 10 を用い，Octane, SunSpider, Kraken の 3 種類のブラウザベンチマークで測定した．提案手法の閾値は，3,000 である．評価結果を**図 8** に示す．図 8 より，オーバヘッドは，2.5% 未満であった．これより，Windows での提案手法のオーバヘッドは小さいことを明らかにした．

4.6　既存手法との比較

提案手法と同じ手法に分類できる DieHarder [18] は，

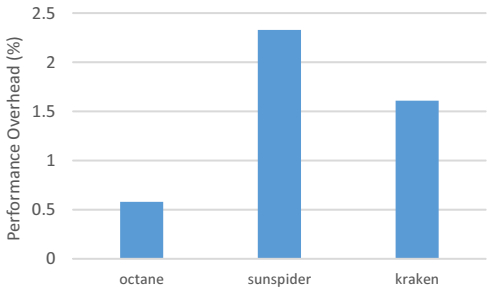

図 8　Overheads of IE10 browser benchmark

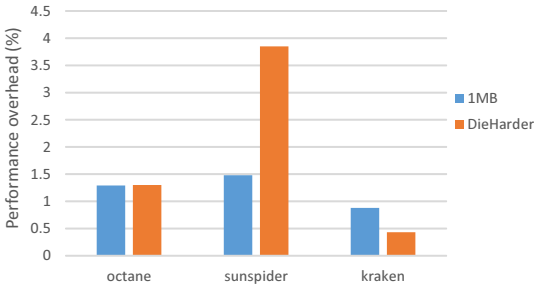

図 9　Overheads of Firefox browser benchmark

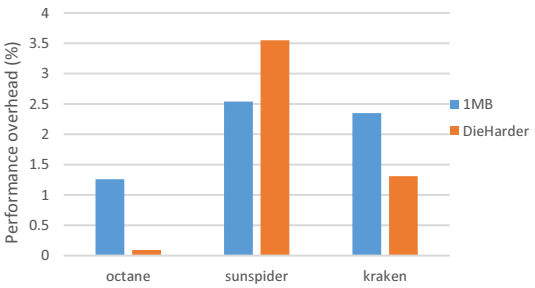

図 10　Overheads of Chrome browser benchmark

ソースコードが公開されているため，提案手法 (閾値 1MB) と比較した．評価では，ブラウザベンチマーク，malloc-test，UnixBench, sysbench, himeno を動作させた場合のオーバヘッドを比較した．また，ブラウザベンチマークを実行した際のメモリ使用量の推移も比較した．

図 9 に，Firefox において，Octane, Sunspider, kraken を実行した際の glibc との性能のオーバヘッドを示している．Firefox では，Kraken 以外では，提案手法の方がオーバヘッドが小さい．また，**図 10** に，Chrome において，Octane, Sunspider, Kraken を実行した際の glibc との性能のオーバヘッドを示している．Chrome では，SunSpider 以外では，提案手法の方がオーバヘッドが大きい．DieHarder は，確保するメモリ領域をある範囲内のメモリ領域からランダムに割り当てる．このとき，同じサイズのメモリ領域を要求すると，解放した領域を即座に再利用されることも多い．また要求するメモリ領域のサイズが大きくなるにつれ，即座に再利用される確率は高くなっている．このため，ある程度大きなメモリ領域を要求するブラウザベンチマークにおいて，DieHarder は，提案手法よりオーバヘッドが小さ

表 3　Overheads of malloc-test.

File size	lib	thread num				
		1	2	3	4	5
512B	1MB	0.437 (17.8%)	0.88 (17.6%)	1.324 (17.1%)	1.765 (16.2%)	2.21 (17.2%)
	DieHarder	1.247 (236%)	2.586 (245%)	4.094 (262%)	5.421 (259%)	6.982 (270%)

表 4　Overheads of UnixBench, sysbench and himeno.

lib	UnixBench (Kbyte/s)	sysbench (s)	himeno (MFLOPS)
1MB	4,130.57 (0.21%)	26.22 (0.24%)	2,688.05 (0.08%)
DieHarder	4,124.77 (0.35%)	26.25 (1.04%)	2,674.44 (0.60%)

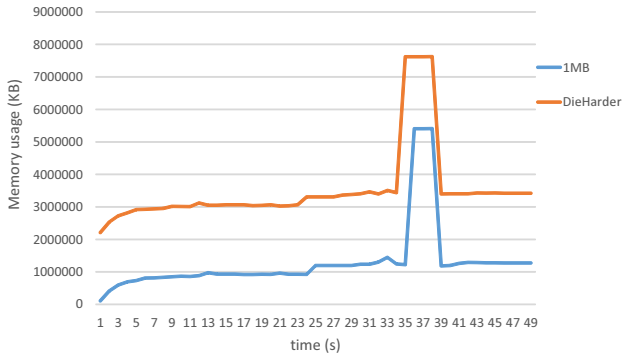

図 11　Overheads of Firefox browser memory usage

くなったと考える．

表 3 に malloc-test で評価した結果を示す．表 3 より，malloc-test では，DieHarder のオーバヘッドが提案手法に比べてかなり大きい．これは，DieHarder の確保するメモリ領域をメモリ空間からランダムに割り当てる処理が原因と考える．このため，オーバヘッドが提案手法に比べ，増加したと推察できる．

表 4 に UnixBench, sysbench, himeno を用い，オリジナルの glibc に対するオーバヘッドを評価した結果を示す．提案手法は，すべてのベンチマークにおいて，DieHarder よりオーバヘッドが小さいことが明らかとなった．

また，**図 11** に，Firefox で Octane を実行した際のメモリ使用量の推移を示している．図 11 より，DieHarder は，提案手法より約 2 倍以上ものメモリ領域を使用していることがわかる．Kraken, SunSpider でも同様の結果となった．

Chrome においては，**図 12** のように，ほぼ同じメモリ使用量の推移となった．しかし，Chrome は，複数プロセスを生成する．そこで，すべての Chrome プロセスの仮想メモリ使用量の合計値を算出し，比較した．合計値は，提案手法では，45,904,020KB であり，Dieharder では，87,906,816KB となった．このことから，Chrome においても，Dieharder は，提案手法より 2 倍近いメモリ使用量を使用していることが分かった．

以上の実験より，ほとんどの場合において，提案手法は DieHarder より処理オーバヘッドが小さく，メモリ使用量が DieHarder より少ないことを明らかにした．また，

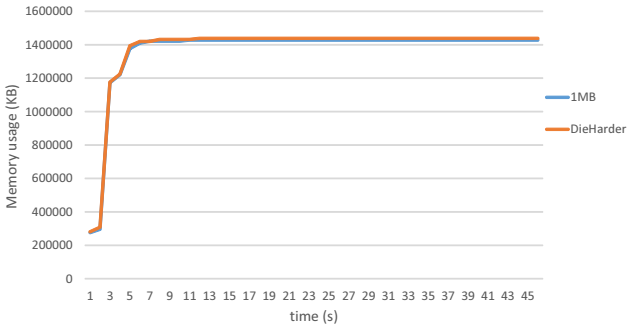

図 12　Overheads of Chrome browser memory usage

DieHarder は，Windows において，適用するプログラムのソースコードが必要であり，ビルド時に DieHarder をコンパイル/リンクする必要がある．一方，提案手法は，ソースコードがないバイナリに対しても適用でき，DieHarder と比べて適用しやすい手法であることがわかる．

5. 考察

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，以下の理由により困難である．

一つ目に，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．

二つ目に，解放するメモリ領域は，前後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の算出を必要とする場合がある．また，前後のメモリ領域と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．

6. おわりに

文献 [1] で提案した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 Linux と Windows で実現し，両 OS での実現方式について述べた．実現したライブラリは，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．

Linux での処理オーバヘッドの評価では，オーバヘッドが小さいことを示した．ただし，メモリの確保と解放を繰

り返す処理では，性能への影響が見られることを示した．また，メモリ使用量の評価では，オリジナルの glibc と比べて，メモリ使用量の増加は小さいことを示した．

次に，Windows について，攻撃コードを用いて UAF 攻撃を行った実験結果を示し，実験したすべての攻撃コードによる攻撃を防げたことを示した．また，ブラウザベンチマークで処理オーバヘッドを測定し，オーバヘッドが 2.5%未満と小さいことを示した．

最後に，既存手法の Dieharder と比較評価を行った．評価の結果，ブラウザベンチマークの結果は，良い場合と悪い場合があったものの，ベンチマークプログラムによる評価では，提案手法の方が処理オーバヘッドが小さいことを示した．また，ほとんどの場合において提案手法の方がメモリ使用量が少ないことを示した．

参考文献

[1] 池上 祐太, 山内 利宏 : 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, コンピュータセキュリティシンポジウム 2014 (CSS2014) 論文集，pp.567–574，2014.
[2] Common vulnerabilities and exposures, https://cve.mitre.org/index.html.
[3] Microsoft Security Intelligence Report Volume 16, <http://www.microsoft.com/en-us/download/details.aspx?id=42646>
[4] 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).
[5] 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).
[6] 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).
[7] 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).
[8] 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)
[9] 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).
[10] Eigler, C.F.: Mudflap: Pointer Use Checking for C/C++, http://gcc.fyxm.net/summit/2003/mudflap.pdf.
[11] 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).
[12] 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).
[13] Pageheap, http://technet.microsoft.com/ja-jp/library/cc835607.aspx.
[14] Electric fence, http://elinux.org/Electric_Fence.
[15] DUMA, http://duma.sourceforge.net/.
[16] Akritidis, P.: Cling: A Memory Allocatorto Mitigate Dangling Pointers, Proc. 19th USENIX Conference on Security, pp.177–192 (2010).
[17] 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).
[18] Novark, G., Berger, E.D.: Dieharder: Securing the heap, Proc. 17th ACM Conference on Computer and Communications Security, pp.573–584, (2010).
[19] 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/.
[20] 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/.
[21] Younan, Y.: FreeSentry: FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers, Proc. 22th Annual Network and Distributed System Security Symposium, NDSS (2015)
[22] Zhang, C., Songz, C., Chen, K.Z., Cheny, Z. and Song, D.: VTint: Protecting Virtual Function Tables' Integrity, Proc. 22th Annual Network and Distributed System Security Symposium, NDSS (2015)
[23] Gawlik, R., Holz, T.: Towards Automated Integrity Protection of. C++ Virtual Function Tables in Binary Programs, Proc. 30th Annual Computer Security Applications Conference, pp.396–405, (2014)
[24] Octane 2.0, <http://octane-benchmark.googlecode.com/svn/latest/index.html>
[25] SunSpider 1.0.2 JavaScript Benchmark, <https://www.webkit.org/perf/sunspider/sunspider.html>
[26] Kraken JavaScript Benchmark (version 1.1), <http://krakenbenchmark.mozilla.org/>
[27] LiteBrite Benchmark, <http://ie.microsoft.com/testdrive/Performance/LiteBrite/>
[28] Peacekeeper, <http://peacekeeper.futuremark.com/>
[29] Dromaeo, <http://dromaeo.com/>
[30] UnixBench, <https://code.google.com/p/byte-unixbench/>
[31] sysbench, <https://launchpad.net/sysbench>
[32] Himeno benchmark, <http://accc.riken.jp/2444.htm>
[33] 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).
[34] Metasploit, <http://www.metasploit.com/>