

Undangle: Early Detection of Dangling Pointers in Use- After-Free and Double-Free Vulnerabilities

Juan Caballero, Gustavo Grieco, Mark Marron, Antonio Nappa

ISSTA'12, July 15–2012, Minneapolis, MN, USA

2017/04/04

概要

- Use-after-free脆弱性(Uaf)及びDouble-free脆弱性(Df)が近年急増
- Uaf及びDfを引き起こす宙ぶらりんポインタを検出、報告
 - 既存研究では宙ぶらりんポインタの使用に焦点
 - 本研究は宙ぶらりんポインタの作成に焦点
 - →早期検出技術(Early Detection)
- 早期検出技術を用いたツール、Undangleの実装
 - 宙ぶらりんポインタの識別法の解説
- Undangleの評価
 - メジャーなWebブラウザを含むアプリケーションにUndangleを適用
 - Firefox, IEなど
 - 開発者が見逃したバグを見つける

Use-after-freeとは？

- 宙ぶらりんポインタ(Dp)がデリファレンスされると生じるバグの一種
 - デリファレンス:ポインタの指す値を見る、取得すること
- 秘匿値を読む、制御フローの改竄、重要データの上書きetc...に悪用可能
 - ゼロディの温床に成り得る
 - ゼロディ:セキュリティホールが周知されてから対策されるまでの期間に攻撃されること
- 通常発生すると何が起きるかは予測不可
 - ここで言う通常は、恶意なしに生じた場合
 - Dpを使うこと自体が未定義動作であるため
 - →プログラム内のデータを書き換えることもあるですよ(あゝ無情)
 - →クラッシュも起きるですよ(生じる現象の中では有情)
- 類似現象にDouble-free(二重解放)がある
 - こちらはDp先を解放しようとすると発生
 - こちらも普通は挙動が予測不可
 - 危険性ではUafに劣るが、起きてほしくないバグ

ようするに
宙ぶらりんが
悪いんや
こゐずみ

Uafの現状

- ・2008年以降急増
- ・2013年には使用される脆弱性No1に
 - 全体の約半分
 - 出典:Security Intelligence Report Vol. 16
- ・特にWebブラウザで頻発
 - Uaf全体の7割がブラウザ上
 - ブラウザではオブジェクトの共有管理が困難なことに起因
- ・代表例:
 - GoogleやAdobeへのオーロラ攻撃
 - 2009～2010
 - IE3へのゼロデイ
 - 2011

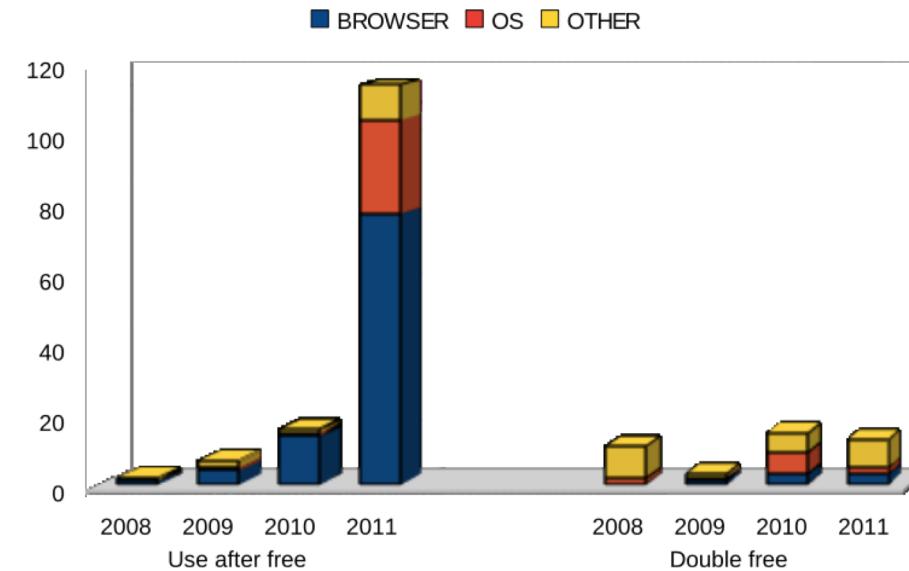
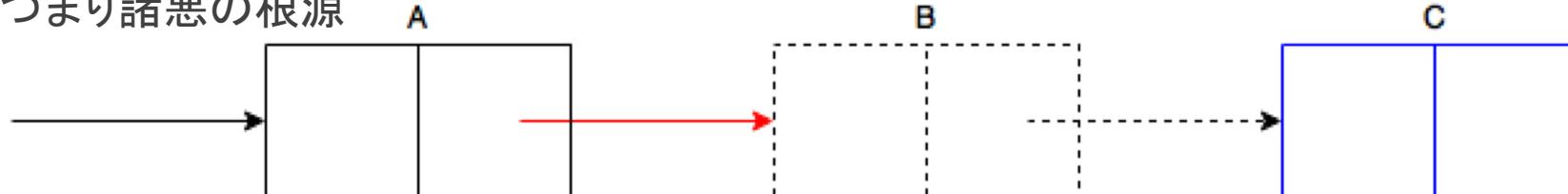


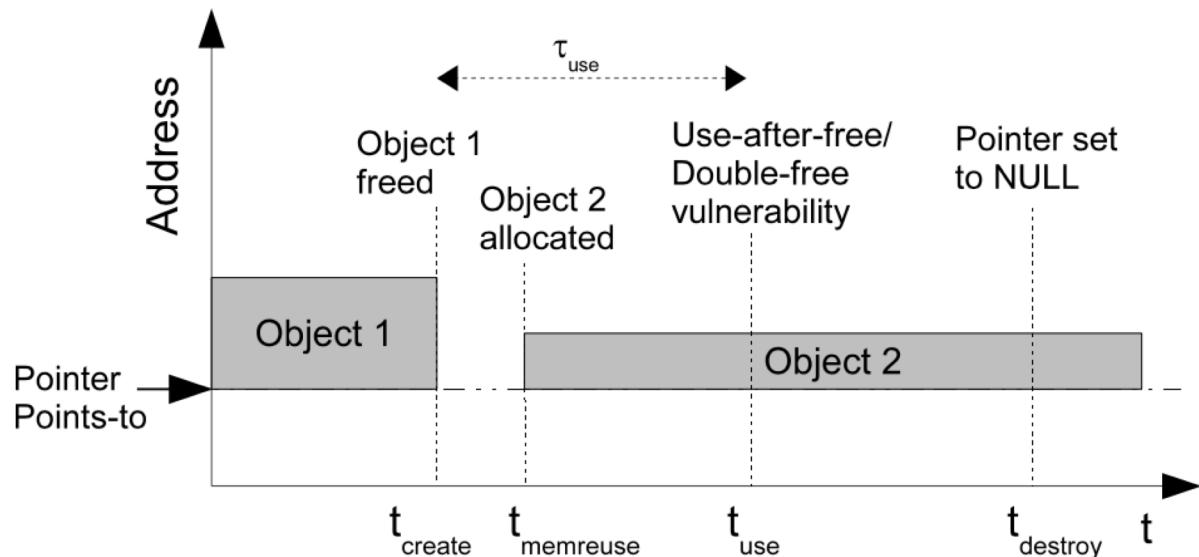
Figure 1: Number of use-after-free (left) and double-free (right) vulnerabilities reported in the CVE database in the 2008-2011 period, split by vulnerabilities in browsers, OSes, and other programs.

宙ぶらりんポインタ(DanglingPointer:Dp)

- ・オブジェクトを解放した際に、そのオブジェクト跡地(デッドメモリ)を指し続けるポインタのこと
 - オブジェクトBが解放された場合の、下図の赤ポインタ
 - ちなみにオブジェクトCはリーク
 - 図の参考:書籍「ガーベージコレクション」著Richard Jones et al
- ・本研究ではポインタを自身のアドレスと、指しているアドレスのペアと表現する(普通?)
 - 下図では矢印の付け根が自身のアドレス、矢先が指しているアドレス
 - 指しているアドレス先が解放されると、ポインタはDpに変化する
 - Dpを元にコピーしたポインタ、Dpをポインタ演算に使用したポインタもDpとする
- ・Dpが使用(デリファレンスもしくは解放渡し)されると、UafとDfが発生
 - つまり諸悪の根源



宙ぶらりんポインタ(Dp)のLifeCycle



- 指しているオブジェクトが解放
 - Dpの生成(t_{create})
- Dpの使用(t_{use})
 - →Uaf(or Df)の発生
 - →ex)悪意のあるコンテンツの実行など
- Dpが上書き、もしくは削除
 - Dpの破棄($t_{destroy}$)
- 使用されずに破棄されれば問題ないが...

Figure 2: Dangling pointer lifetime.

宙ぶらりんポインタ(Dp)の分別

- Dpは以下のように分別する

- 危険(unsafe):プログラム実行パスの内、一度でも使用されるDp
 - 一刻も早く修正すべき対象
- 潜在的に危険:作成されるが、プログラム実行中は決して使用されないDp
 - 直ちに影響はないが、パッチなどによって将来的に使われる可能性がある
 - やはり修正すべき対象
- 安全:作成されるが、使われず正しく破棄されるDp
 - Dp自体はコンパイル時にも生じるし、プログラマが意図的に作れる
 - ただし、これらは基本的に短命である

- Dpが生じるバグの種類の分別

- 非共有バグ:オブジェクト解放によって生じるDpが一つ
- 共有バグ:オブジェクト解放によって生じるDpが二つ以上
- 一般に、共有バグでのDpの識別と修正は非共有より困難
 - 判定にオブジェクトタイプを考慮する必要性

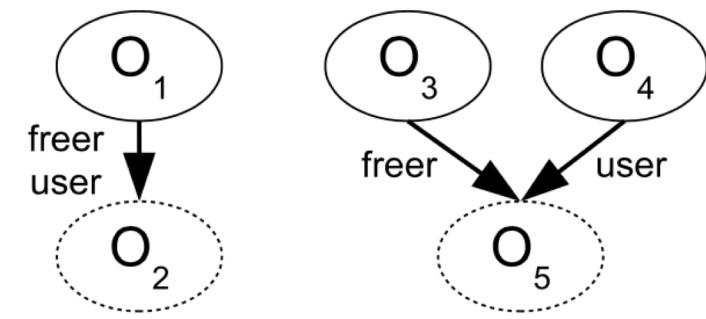


Figure 3: Classes of dangling pointer bugs. The dashed object is deallocated creating the dangling pointers.

既存研究との差異

- ・既存研究では、宙ぶらりんポインタ(Dp)の使用に焦点
 - 先のt_use
 - Late Detection(晚期検出)
- ・本研究では、Dpの作成に焦点
 - 先のt_create
 - Early Detection(早期検出)
 - 新しい切り口としての提案
- ・結果として、今まで発見できていなかった脆弱性の発見@Firefox
 - →前述の潜在的なDpの検出ができる点で既存研究より優れていると言える

早期検出(Early Detection)

- ・前提: アプリもしくはパッチのテスト段階で使用する
- ・危険なDpの判定のアルゴリズムの一つ
 - 長生きなDpは危険という直感からの発想(=安全なDpは短命)
- ・潜在的、危険なDpは長命という特徴を持つという直感
 - →特定の閾値(**Safety window**サイズ)を超えるDpは危険と判断
 - 潜在的なDpを判定する理由
 - 修正のためのパッチが、潜在的なDpを**危険なDpに昇華させる可能性**があるから
 - →ゼロデイ発生の危険性
- ・このアルゴリズムの肝
 - 危険かどうかの適切な閾値(**Sw**サイズ)を決める
 - **宙ぶらりん**を正しく判定する
 - ここが肝中の肝

Safety window(Sw)

- Dpが生じた段階でセットされ、特定の閾値(サイズ)でトリガされる
 - →Swがトリガされた段階で生きているDpは長命＝危険と判定
- Swが持つ要素は2つ
 - ①**サイズ**:単位は命令数、例えば1000が入れられれば、Dp生成から1000命令後という意味
 - ②生成時刻:Dpが生成された時刻
- SwはDpごとに割り当てられる
 - そしてサイズのカウントはスレッド単位で進行する
 - =別のスレッドの命令はカウントしない
- 今回の実験では
 - 脆弱性解析→サイズ無限
 - アプリテスト用→サイズ5000

テストへの入力と出力

- ・入力:外部ツールで入力を作成
 - 多種多様な入力例を生成するため
 - 本研究ではbf3(Browser Fuzzer 3)を使用
 - 他でも代用可
- ・早期検出での出力
 - 前提:Dpが危険とフラグされたor使用された
 - ①使用されたなら、生じた脆弱性の種類(Uaf or Df)
 - ②解放済みバッファの情報
 - ③オブジェクトを解放したスレッドと、Dpを使用したスレッド
 - ④プログラムのコールスタック
 - ⑤生きているDpのリスト
 - ⑥Dpをリストしているバッファの情報

Undangle

- ・早期検出の技術を用いたDp検出ツール(動作はオフライン)
 - 入力に割り当てログ(Allocation log)と実行トレース(Execution trace)が必要
- ・デバッグツールとしては汎用性に難があるが、UafとDfの判定に特化している
 - クラッシュや悪用の原因がUafやDfであれば、その情報を自動で収集してくれる
- ・脆弱性解析での利点
 - より多くの情報を自動で収集
 - 既存技術より詳細な情報を出力
 - 潜在的な危険を持つDpを識別可能

Undangleの挙動

- ・実行モニタを通して割り当てログと、実行トレースを得る
- ・→ログとトレースをUndangleに入力する
- ・Undangle内では3つのモジュールが動作
 - テイント追跡モジュール(**TTモジュ**)
 - テイント:汚れ。ここではマークされたポインタを指す
 - ポインタ追跡モジュール(**PTモジュ**)
 - 宙ぶらりん検出モジュール(**DDモジュ**)
- ・→最終的にDDモジュが診断情報を出力

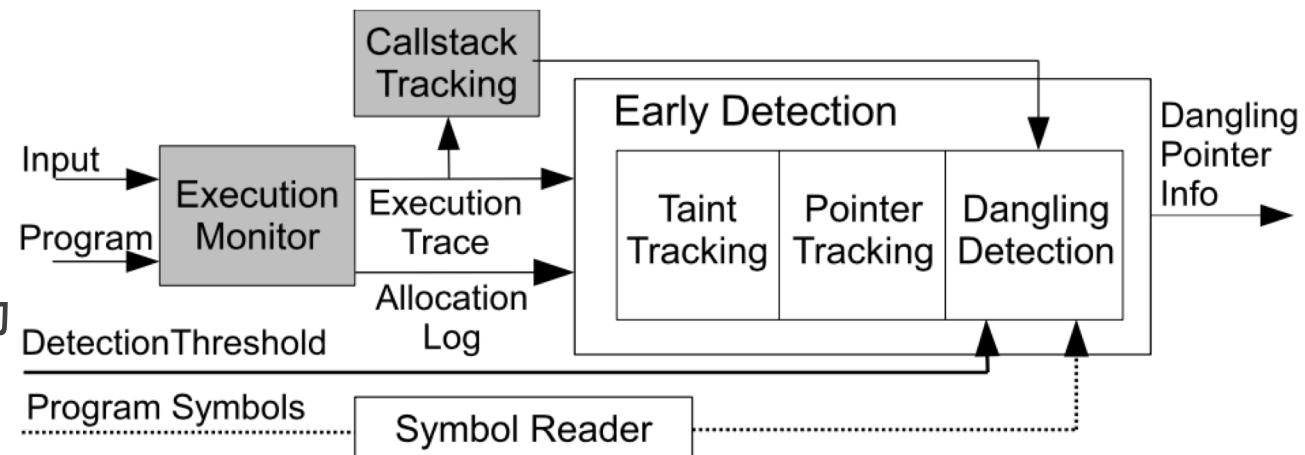


Figure 4: Architecture Overview. Gray boxes were previously available.

汚染追跡モジュール(TTモジュ)

- ・既存の汚染追跡技術はforward mapを用いている
 - Forward map: ここでは、オブジェクト内のポインタがどこを指しているかの関連付け
- ・本研究で実装したTTモジュはforward mapに加えて、reverse mapも採用
 - Reverse map: ここでは、オブジェクトを指しているポインタはどこから来ているかの関連付け
 - 解放されるオブジェクトのreverse mapがDpを表現
 - DDモジュで使用

ポインタ追跡モジュール(PTモジュ)

・宙ぶらりん(dangling)を型として扱える独自の推論モジュール

・非宙ぶらりんポインタの判定

- ルートポインタ(root pointer)の判定

- →他のポインタから導出できないポインタ
- →関数戻り値や、コールスタックから取得可能
- →ルートポインタから辿れるポインタにテイントを施す

T
↓
Ptr
↓
Dangling
↓
⊥

・テイント済みポインタでのコピー、ポインタ演算から導出されるポインタの判定

- ポインタを新規に生成する特定の命令(コピー、ポインタ演算など)の際に、新たにできるポインタをテイント

・導出されるポインタにはテイントラベルを伝搬(右図)

- ルートポインタの情報、オフセットなど

・要するに、トレースの各命令で使用されたポインタをテイントしていく

```
type taint_label = {  
    type : Ptr | Dangling  
    root-type: HeapRoot |  
    StackRoot |  
    [...] |  
    Pseudo-root  
    root-addr : int64;  
    icounter : int64;  
    offset : int;  
}
```

Figure 5: Lattice
of pointer types.

Figure 6: Pointer taint label.

PTモジュ:ポインタの導出

Table 1: The x86 instructions that propagate pointers, their abstraction, and the associated propagation rules.

Instructions	Abstraction	Rules
mov, movs, push, pop	$dst \leftarrow src$	move
xchg	$t \leftarrow src; src \leftarrow dst; dst \leftarrow t$	exchange
add	$dst \leftarrow src_1 + src_2$	add, default
inc	$srcdst \leftarrow srcdst + 1$	nop, default
sub	$dst \leftarrow src_1 - src_2$	sub, default
dec	$srcdst \leftarrow srcdst - 1$	nop, default
lea	$dst \leftarrow (disp + index) + base$	add, default
$nop^*{}^4$	-	nop
All other	$dst \leftarrow \top$	default

・実行トレースによる動作の問題点

- ①:オペランドに入る値が未実行
- ②:ユーザレベルの命令しかない
 - カーネルにポインタを渡してもポインタが帰ってこない

・①の対処法

- 左図のルールに基いて、ポインタを導出(エミュ)

・②の対処法

- トレース内のオペランドとトラッキングの結果を比較
 - →異なれば、テイントを落として、再デリファレンスを待つ

宙ぶらりん検出モジュール(DDモジュ)

- 以下の機能を持つ

- Dpの生成の識別←TTモジュが必要(逆マップを得るため)
- Swがトリガされた段階でのDpの検出←PTモジュが必要(Dpが生きているかどうかの判定)
 - 危険、もしくは潜在的に危険なDpの検出
- Swがトリガされる前に使用されたDpの検出
 - 文句なしに危険なDpの検出
- 検出されたDpの診断情報の出力

- 割り当てログを基に、解放されたオブジェクト内のポインタを破棄

- →次に、解放objを指していたポインタをDpと判定
 - このために逆マップのリストが必要→リストの中身をDpに指定
 - →Swを設定
- →各命令ごとにSwがトリガされるか判定
 - トリガされた場合、PTモジュを介してDpのリストを得て、リストの中身をフラグ(空ならフラグ無し)
- →診断情報の出力

評価

- 評価は二段構え

- ①本当に脆弱性の使用を判定できるか
 - 実際に脆弱性を引き起こすと分かっているプログラムにUndangleを適用(表2の8つのプログラム)
 - 脆弱性を引き起こすDpの使用を検出するため、Swサイズは無限に設定
- ②二つのFirefoxのバージョンに対してUndangleでテスト
 - Swサイズを決めて、危険or潜在的なDpの判定を行う

- uaf-h:ヒープオブジェクト解放で生じるUaf
- uaf-m:メモリ内のライブラリのアンロードで発生する特殊なUaf
- dfree: double-free
- uf: under-flow
 - ※ie7-uf: CVE上ではUafと報告されているが誤り
 - ※実際はufで、UndangleではUafは検出されず

Table 2: Vulnerabilities used in the evaluation.

Name	Program	Vuln. CVE	Type
apache	Apache 2.2.14 mod_isapi	2010-0425	uaf-m
aurora	Internet Explorer 7.0.5	2010-0249	uaf-h
firefox1	Firefox 3.6.16	2011-0065	uaf-h
firefox2	Firefox 3.6.16	2011-0070	dfree
firefox3	Firefox 3.5.1	2011-0073	uaf-h
ie8	Internet Explorer 8.0.6	2011-1260	uaf-h
safari	Safari 4.0.5	2010-1939	uaf-h
ie7-uf	Internet Explorer 7.0.5	2010-3962	uf

評価1の結果

Table 3: Vulnerability analysis results.

Name	Num uses	Threads (inv./exist)	τ_{use}	Dangling (creation)	Dangling (use)	Heap objects	Class	Trace size	Time (sec)
apache	1	1 / 3	767,014	10 (5/4/1/0)	3 (2/0/1/0)	2 / 1	NS	96.8 MB	39
aurora	1	1 / 11	6,938,180	7 (2/4/0/1)	1 (1/0/0/0)	2 / 1	S	1.0 GB	438
firefox1	1	1 / 9	987,707	4 (2/2/0/0)	1 (1/0/0/0)	2 / 1	S	2.0 GB	1,072
firefox2	1	1 / 10	5,364	2 (1/1/0/0)	1 (1/0/0/0)	1 / 1	NS	3.3 GB	1,918
firefox3	1	1 / 7	11,000	5 (2/3/0/0)	1 (1/0/0/0)	2 / 1	S	2.4 GB	1,982
ie8	3	1 / 10	1,984,815	10 (2/7/0/1)	1 (1/0/0/0)	2 / 1	O	0.4 GB	165
safari	1	1 / 6	121,284	8 (3/4/0/1)	3 (2/1/0/0)	2 / 1	NS	0.6 GB	260

評価1、表3の説明

- ・結論:それぞれのテストで、脆弱性の発生を検出
 - Firefox2のみがDfで、他6つはUaf(そしてクラッシュ)
 - Ie8は3回Dpの使用が認められたが、クラッシュは3回目の使用でのみ生じた
 - →正しく検出できている
- ・「Thread」列はDpに(**関与したスレッド数/全体のスレッド数**)を示している
 - →マルチスレッドに関わらず、单一スレッドがDpの作成と使用を引き起こしている証左
 - →マルチスレッドのバグは悪用しづらい?
- ・「τuse」列は、Dpが作成から使用されるまでのスパン(単位は命令数)
 - いかに悪用されるDpが長命かわかる
 - →最長で700万命令後に使用される
 - →早期検出の意義にも繋がる

評価1、表3の説明

- ・2つのDanglingは(脆弱性を引き起こす)Dpが生成(creation)、使用(use)されたときのDpの数
 - それぞれ(ヒープ/スタック/データ/レジスタ)の数
 - 脆弱性を引き起こすDp自体も含む
 - →脆弱性を引き起こすDp以外は短命であることは明白
 - スタック、レジスタ内のDpは特に短命
 - ApacheとSafariは例外
- ・「Heap」列:ヒープ領域内の長命のDpについて焦点を当てたときの、ヒープオブジェクトの数
 - (作成時/使用時)
 - 例えば(2/1)なら、長命Dp生成時に、Dpを持つヒープオブジェクトが2つ、使用時にDpを持つヒープオブジェクトは一つという意味
 - 「class」列:バグの種類
 - NS:非共有、S:共有、O:その他
- ・残り2列は、解析にかかった時間と、トレースのサイズ
 - 時間は1分～33分に収まっている

ケーススタディ

- ・表2のfirefox1とfirefox3のバグを分析
 - 構造としては右図(図7,8)の様に類似性が見られる
 - 結論としては、このバグは同一のモノ
- ・nsXPCWrappedJSオブジェクトがDpを生成し、もう片方が使用
 - nsXPCWrappedJS側のDpは短時間で破棄
- ・バージョンによって行番号が異なるが、同一箇所がバグの原因
 - →パッチを当てたが修正されていなかった部分ということ
 - →現在は修正済み(firefox6.0.2以降)

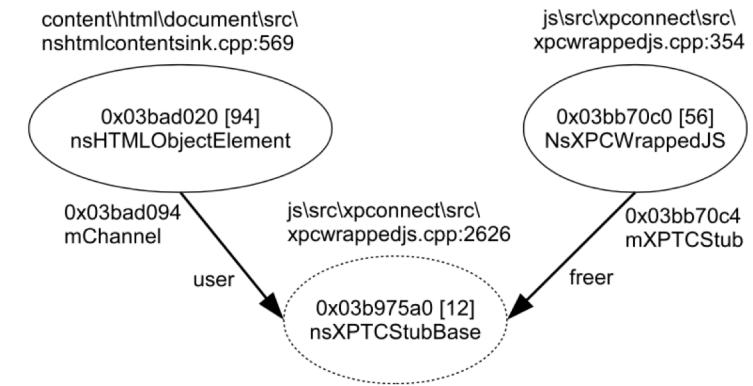


Figure 7: Relevant heap state at creation for the firefox1 vulnerability (Firefox 3.6.16).

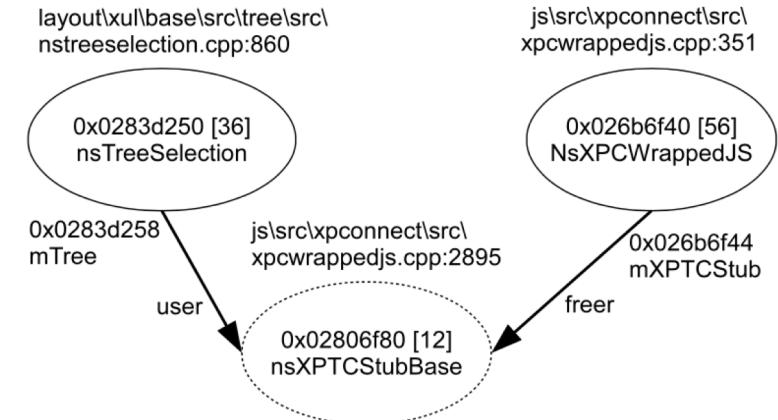


Figure 8: Relevant heap state at creation for the firefox3 vulnerability (Firefox 3.5.1).

Undangleのテスト評価

- 適切なSwサイズの評価
 - →長すぎると、危険なDpが実行されてしまう可能性
 - →短すぎると安全なDpも削除して挙動がおかしくなる可能性
 - →安全なDpを削除しない最小のSwサイズが望ましい
- Undangleが見つけた(危険な)Dpを手動で解析
- 実際に適用してテスト@Firefox10.0

Sw評価

- ・1列目はDpの数
- ・2列目はDpを生むバグの数(当然Dp数 \geq バグ数)
- ・3列目は、バグの類似性を消去した数
- ・4列目は、共有バグの数
- ・共有バグの数を減らさずに、アラームの減少が最大かつ表3のtuseの最小値5436を下回る値
 - → 500→5000の間がベストポジ →5000に大決定

手動で確かめたべや

- ・12個の共有バグについて
 - 共有バグが一番既存手法では判定が難しい=本研究のモチベ
 - 4つがユニーク、残り8つはその4つのいずれかの複製
 - →3/4が潜在的、1/4が危険
 - →3/4については取り急ぎ危険ではないが、今後のパッチ如何では危険に成り得るので、修正すべき
 - →1/4については、Mozillaに報告 →バグ認定

実際にUndangleをテストしてみた

- ・環境:Firefox10.0(当時の最新バージョン)
- ・入力:外部ツールから計30個(JSファジー10,DOMファジー10,XMLファジー10)
- ・結果:それまで未発見の非共有バグ2件を発見
 - 誤認率は0
 - 結果から5分でどのポインタでバグが生じているか理解できた(Firefox専門知識がない著者の1人が)

Table 5: Testing results on 30 inputs generated by the Bf3 web browser fuzzer.

Window size	Fuzz mode	Num traces	Num Dang.	Num bugs	Stack hash	Sharing bugs
5,000	JS	10	30	30	2	0
5,000	DOM	10	30	30	2	0
5,000	XML	10	30	30	2	0
5,000	All	30	90	90	2	0

結論

- ・宙ぶらりんポインタへの焦点を**使用(use)**から**作成(create)**にシフトしたアプローチの提案
- ・Dpの作成を検出するモジュールの開発、実装
 - →Undangle
- ・Undangleを評価すると、UafとDfの原因の究明により役立つことが示された
 - これまで検出が難しかった、**潜在的危険を孕むDp**の判定に成功