機械語命令の類似度算出による 複製された脆弱性の発見手法の提案

中島 明日香† 岩村 誠† 矢田 健†

†ＮＴＴセキュアプラットフォーム研究所
〒180-8585 東京都武蔵野市緑町 3-9-11
{nakajima.asuka, iwamura.makoto, yada.takeshi}@lab.ntt.co.jp

あらまし サイバー攻撃の原因の一つとしてソフトウェア中に存在する脆弱性が挙げられる．その中でも，ソフトウェア開発過程において脆弱性部分のソースコードが複製される事(コードクローン)により発生する脆弱性がある．本研究では実行ファイルを対象に，コードクローンにより生じた脆弱性の発見手法を提案する．具体的には，過去に発見された脆弱性部分から機械語命令を抽出しオペランドを正規化した後，類似文字列検索アルゴリズムを用いて複製先の脆弱性を発見する．提案手法の有効性を示すため，実在のコードクローンの脆弱性を利用し脆弱性の複製元箇所と複製先ソフトウェア間との類似度算出を行った結果，最低でも60.7%の類似度が算出された．また，本提案手法を用いて実行ファイル40945個に対して検査を実施した所，実際に過去コードクローンの脆弱性であったものを発見した．

A Detection Method of Reproduced Vulnerability based on Similarity of Machine Code Instruction

Asuka Nakajima† Makoto Iwamura† Takeshi Yada†

†NTT Secure Platform Laboratories
3-9-11 Midori-Cho, Musashino-Shi, Tokyo, 180-8585 JAPAN

{nakajima.asuka, iwamura.makoto, yada.takeshi}@lab.ntt.co.jp

Abstract Nowadays, one of the root causes of cyber-attacks is software vulnerability. Therefore, in this research, we propose a method that can detect reproduced vulnerability, which is caused by the reused vulnerable source code (code clone), in executable file. To detect reproduced vulnerability, first we normalized the operands of a machine code instruction which extracted from vulnerable part of the software, and then search the similar part in the target software by using approximate string search algorithm. To verify the validity of the proposed method, we used code clone vulnerability, which was found in real world software, and calculate the similarity between the source of code clone vulnerability and the software which contains reproduced vulnerability. As a result, the calculated similarity was 60.7% at minimum. Moreover, we audited 40945 executable files, by using our proposed method, and we found real world code clone vulnerability, which was truly vulnerable in the past.

1 研究の背景と動機

マルウェア感染をはじめとしたサイバー攻撃の中には, ソフトウェア中に存在する脆弱性を悪用して行われているものがある. 脆弱性を利用した攻撃を緩和する, データ実行防止などの技術は存在するものの, 回避手法が存在するため攻撃を完全に防ぐことは難しい. そのため, 脆弱性が発見された場合, 出来る限り早期の修正が必要とされている.

脆弱性がもたらす脅威と, 修正の必要性についてはソフトウェアベンダーの中でも広く認知されており，主要製品に限れば 8 割近くの脆弱性が，その

脆弱性情報公開前後 7 日以内に修正パッチが配布されている[1]．しかし，脆弱性の中でも複数の製品に跨るような場合，その修正パッチ配布において下記二点の問題が生じることがある．

1）製品間で修正パッチが配布される時期に差が生じる問題
2）複数製品の内の一部の製品のみ，その脆弱性の修正パッチが配布される問題

複数製品間に跨る脆弱性の修正時に上記が生じた場合，攻撃者は配布された修正パッチを解析して得られた情報を基に，未修正のソフトウェアに対する攻撃コードを作成し，攻撃することが可能になる．

過去の調査[1]では，脆弱性が 2 つのアプリケーションに跨っていた場合，片方の脆弱性が修正されてから，もう片方脆弱性が修正されるまでの期間は最大で 118 日かかるという結果が出ており，攻撃者が修正パッチを解析し，攻撃するまで十分な猶予があると言える．さらに，そもそもソフトウェアベンダーの調査不足で，一部製品のみの脆弱性しか修正されない場合，ユーザが無防備な状態であるにも関わらず，攻撃側は時間的制約に縛られず修正パッチを解析し，自由に攻撃する事が可能となる．実際に Firefox に発見された脆弱性と同一の脆弱性が Thunderbird に存在していたにも関わらず，見逃されていたといった事例も発見されている[2]．

複数製品に脆弱性が跨る原因としては，ソフトウェア開発時に他のソフトウェアのソースコードから，脆弱性部分を含めたソースコードが複製(コードクローン)されることが挙げられる．ソフトウェアベンダー内の複数製品間で脆弱性が複製される場合もあれば，オープンソースソフトウェアからプロプライエタリなソフトウェア中に脆弱性が複製される場合もある．もしプロプライエタリなソフトウェアに脆弱性が複製された場合，ソースコードが簡単には入手出来ない為，その発見が難しくなる．

そこで本稿では実行ファイルを対象に，コードクローンにより生じた脆弱性を，機械語命令列間の類似度を算出する事により発見する手法を提案する．

2　従来研究とその課題

コードクローンの脆弱性発見手法は，ソースコードを対象としたもの[2][3][4]と，実行ファイルを対象としたもの[5][6]に大別できる．

ソースコードを対象としたコードクローンの脆弱性発見の研究は，ソースコード間のコードクローン探索の研究[7][8]の応用技術に位置する．例えば，Jang らが提案した ReDeBug[3]や Li らの提案した手法[4]では，修正パッチを基に脆弱性箇所を含むソースコードを抽出した後，ソースコードを正規化・トークン化し，同様に処理した検査対象ソースコードから同一箇所を探索する事により，コードクローンの脆弱性を発見する．

上記研究では，ソースコードを対象としているため，その手法をそのまま実行ファイルを対象としたコードクローンの脆弱性探索に利用する事は難しいと言える．

実行ファイルを対象としたコードクローンの脆弱性発見技術としては，まず Pewny らの TEDEM[5]が挙げられる．TEDEM では，逆アセンブルした脆弱性箇所をベーシックブロック単位に分割した後，そのベーシックブロックに含まれている機械語命令列を中間表現に変換し，その後式木として表現した上で，同様に表現した検査対象プログラムのプログラムコード間との，木の編集距離を算出する事で，コードクローンの脆弱性箇所を発見する．この手法では，コンパイル環境の違いによる変化には対応出来るが，ソースコードが複製される際に追加や削除が行われるなどの大幅な変化が発生した場合には，脆弱性の検出が難しくなると考えられる．

同じく Pewny らは，上記の他にも実行ファイルの対象アーキテクチャの違いを考慮した，コードクローンの脆弱性発見技術[6]を提案している．この手法では，まず脆弱性箇所をベーシックブロックに分割し，そのベーシックブロックに対して，複数回ランダムな入力値を与え，入力値とその出力値の結果のペアを，MinHash を用いてハッシュ化する．そして検査対象プログラム中のベーシックブロックにも同様の処理をした上で，ハッシュ値同士を比較する事でコードクローンの脆弱性箇所を発見する．この手法では，ベーシックブロックの入出力値を基にしているため，複製先ソースコードに追加や削除が発生した場合，入出力値が変化してしまう可能性がある．そのため前述の研究と同じく，この場合では脆弱性の検出が難しくなると考えられる．

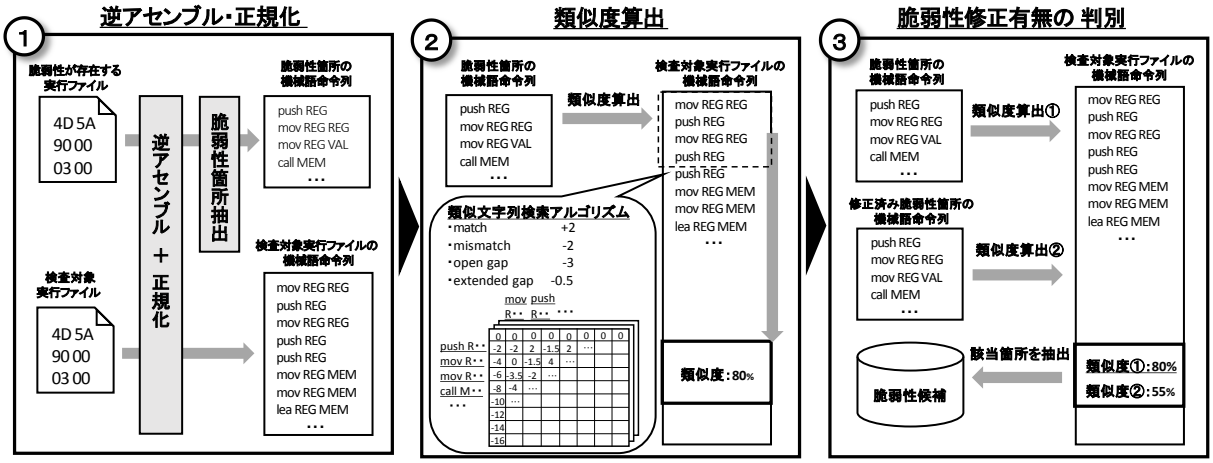

図 1：提案手法概要

3 提案手法

前述のように，実行ファイルを対象としたコードクローンの脆弱性発見手法は存在するものの，複製先脆弱性箇所のソースコードに追加・修正があった場合，その検知が難しくなると考えられる手法が提案されていた．

そこで本論文では，実行ファイルを対象に，複製先脆弱性箇所がソースコードレベルで一部改変されていた場合でもコードクローンの脆弱性として発見可能な手法を提案する．

3.1 提案手法概要

本提案手法の全体像を図 1 に示す．本提案手法では，まず既知の脆弱性箇所の機械語命令列を正規化したのち，同様に正規化された検査対象ソフトウェアの機械語命令列中から，比較文字列間の文字列の追加・削除にも対応した類似文字列検索アルゴリズムを用いて脆弱性箇所との類似度を算出する．次に，類似度が最大となる部分が一定の閾値を超えていた場合，今度は事前に用意した修正済み脆弱性箇所との類似度も算出し，該当箇所が修正済みの脆弱性か否かを判断する．そして修正済み脆弱性箇所との類似度が最初に算出された類似度より低い場合，未修正の複製先脆弱性候補として該当箇所を抽出する．各手法の詳細について次項以降述べる．

3.2 正規化

ソースコードから実行ファイルを生成する際に，そのコンパイル環境によって，同じソースコードでも異なる機械語が生成される事がある．具体的にはオペランド部分において，利用されるレジスタやメモリアドレス，そして即値が変化する．

そこでオペランド部分で利用されている，レジスタ・メモリアドレス・即値をその特徴を表した文字列に変換する正規化処理を行う．具体的には，レジスタを「REG」，メモリアドレスを「MEM」，即値を「VAL」という表記に変換する．このように正規化する事でコンパイル環境の違いによるオペランドの変化に対応できる．

3.3 類似度算出

本項では，正規化された脆弱性箇所と検査対象ソフトウェアの機械語命令列間の類似度を算出する具体的な手法について述べる．

まず，正規化済みの脆弱性箇所の機械語命令列を A，正規化済みの検査対象ソフトウェアの機械語命令列を B とすると，B の中で A と類似した箇所を，スコアを基にした類似度を求めることで特定する．ここでは $|A| = M$，$|B| = N$ とし，$A = a_1^M = a_1, a_2, a_3 \ldots a_M$, $B = b_1^N = b_1, b_2, b_3 \ldots b_N$ とする．スコアは，動的計画法に基づいた類似文字列検索アルゴリズムである Needleman-Wunsch[9]に，文字列の追加・削除部分中の位置に応じて減点を区別するアフィンギャップペナルティ[10]と呼ばれる手法を適用し，さらにスコア計算部分を変更する事によって算出できる．これにより複製先脆弱性箇所のソースコードの一部に，複数行に渡る追加や削除があったとしても類似度の低下を抑えられる．

A,B 間の算出スコアを $F(A,B)$ とすると $\frac{F(A,B)}{F(A,A)}$ にて類似度を算出する事ができる．スコア計算のための具

体的な処理内容について説明する. まず *A, B* 間で, 3 つのスコア行列 $X=\{x_{ij}|1\le i\le M, 1\le j\le N\}$, $Y=\{y_{ij}|1\le i\le M, 1\le j\le N\}$, $Z=\{z_{ij}|1\le i\le M, 1\le j\le N\}$ の各要素を下記の式で算出する. *match*, *mismatch* のスコアは任意に設定できる. o はギャップ(追加・削除)の開始スコア(*opengap*)とし, e はギャップの継続スコア(*extendedgap*)とする. ここでは o と e のスコアは任意に設定できる.

$$x_{ij}=\begin{cases}0 & if\ i=0\ and\ j\neq 0,\\ i\times mismatch & if\ j=0,\\ \boldsymbol{max}\begin{cases}x_{i-1,j-1}+s(a_i,b_j)\\ y_{i,j}\\ z_{i,j}\end{cases} & otherwise.\end{cases}$$

$$s(a_i,b_j)=\begin{cases}match & if\ a_i=b_j,\\ mismatch & otherwise.\end{cases}$$

$$y_{ij}=\begin{cases}-\infty & if\ i=0,\\ 0 & if\ j=0\ and\ i\neq 0,\\ \boldsymbol{max}\begin{cases}y_{i-1,j}+e\\ x_{i-1,j}+o+e\end{cases} & otherwise.\end{cases}$$

$$z_{ij}=\begin{cases}0 & if\ i=0\ and\ j\neq 0,\\ -\infty & if\ j=0,\\ \boldsymbol{max}\begin{cases}z_{i,j-1}+e\\ x_{i,j-1}+o+e\end{cases} & otherwise.\end{cases}$$

X は *A,B* 間の *match, mismatch* スコアを管理する行列であり, Y と Z はそれぞれ, *A, B* におけるギャップスコアを管理する行列である. 上記 3 つのスコア行列を利用して, 下記計算式で得られる最大スコア点 j_{max} を基に類似度 $\frac{F(A,B)}{F(A,A)}$ を算出する.

$$j_{max}=\underset{1\le j\le N}{\operatorname{argmax}}\, x_{Mj}$$

3.4 脆弱性修正有無の判別

検査対象ソフトウェア中で, 脆弱性箇所との類似度が最大となる箇所が一定の閾値を超えていた場合であっても, 該当箇所が脆弱性であるとは限らない. なぜならば, 既に修正済みの複製先脆弱性箇所である可能性があるからである. そのため, 該当箇所の脆弱性修正の有無を判別する必要がある.

そこで本手法では, 検査対象ソフトウェア中に, 脆弱性箇所と高い類似度が算出された箇所が発見された場合, 今度は修正済み脆弱性箇所を利用して類似度を算出し, その類似度が最初の類似度よりも低い場合, 未修正の複製先脆弱性箇所とする. そして該当箇所を脆弱性箇所候補として抽出する. 該当箇所は, 類似度算出部分で算出した最大スコア点 j_{max} を起点に, j_{max} 算出までに至った, 行列の算出式の中の要素選択順序を逆順に, i=1 になるまで辿っていくことで抽出する事が出来る.

4 実験

本章では提案手法の有効性を示すために行った二つの実験について述べる.

4.1 実験 1

実験 1 では, 提案手法を用いて算出された類似度に基づき, 複製先が一部改変済みであるコードクローンの脆弱性が実際に特定可能か否かを評価する実験を行った. 具体的には脆弱性が複数製品に跨っており, かつ複製先脆弱性箇所が一部改変されている事が判明済みのものを利用して下記を実施した.

1) 脆弱性の複製元箇所と, 複製先ソフトウェア間との類似度算出
2) 脆弱性の複製元箇所と, コードクローンの脆弱性が存在していないと考えられる実行ファイルのデータセット間との類似度算出

実験に利用した各脆弱性(CVE-2008-4316・CVE-2008-5023)とデータセットの詳細について述べる.

- CVE-2008-4316

本脆弱性は glib(ver2.20 未満)と呼ばれるライブラリ中にある g_base64_encode 関数に存在する整数オーバーフローの脆弱性である. 暗号鍵管理ソフトウェアの seahorse1.01 にて同一脆弱性箇所が発見されている事が知られている. 脆弱性の複製元となった glib のソースコードを図 2 に, 複製先となった seahorse のソースコードを図 3 に示す.

```
g_base64_encode (const guchar *data,gsize len){
 gchar *out;
 gint state = 0, outlen;
 gint save = 0;

 g_return_val_if_fail (data != NULL, NULL);
 g_return_val_if_fail (len > 0, NULL);

 out = g_malloc (len * 4 / 3 + 4);
 outlen = g_base64_encode_step (data, len, FALSE, out, &state, &save);
 outlen += g_base64_encode_close (FALSE, out + outlen, &state, &save);
 out[outlen] = '¥0';

 return (gchar *) out;
}
```

図 2: glib2.19 のソースコード(CVE-2008-4316)

```
seahorse_base64_encode (const guchar *data,gsize len){
  gchar *out;
  gint state = 0, outlen;
  gint save = 0;

  out = g_malloc (len * 4 / 3 + 4);
  outlen = seahorse_base64_encode_step (data, len, FALSE, out, &state, &save);
  outlen += seahorse_base64_encode_close (FALSE,out + outlen,&state,&save);
  out[outlen] = '¥0';
  return (gchar *) out;
}
```

図 3: seahorse1.01 のソースコード(CVE-2008-4316)

図 3 から, 複製先では g_return_val_if_fail 関数に関係する部分が削除されている事が分かる.

- CVE-2008-5023

本脆弱性は Firefox と Seamonkey の中に含まれる AllowScripts 関数に存在する脆弱性である．具体的にはセキュリティチェックが不十分なため，署名済み jar ファイルに任意のスクリプトが挿入可能な脆弱性である．脆弱性の複製元であると考えられる Firefox のソースコードを図 4 に，複製先と考えられる Seamonkey のソースコードを図 5 に示す．複製元と先の大半は同じだが，複製先では一部ソースコードが変更・追加されている事が判明している．

```
PRBool nsXBLBinding::AllowScripts (){
  PRBool result;
  mPrototypeBinding->GetAllowScripts(&result);
  ...
  JSContext* cx = (JSContext*) context->GetNativeContext();

  nsCOMPtr<nsIDocument> ourDocument;
  mPrototypeBinding->XBLDocumentInfo()->GetDocument(getter_AddRefs(ourDocument));
  PRBool canExecute;
  nsresult rv = mgr->CanExecuteScripts(cx, ourDocument->NodePrincipal(), &canExecute);
  return NS_SUCCEEDED(rv) && canExecute;
}
```

図 4: Firfox3.0.1 のソースコード(CVE-2008-5023)

```
PRBool nsXBLBinding::AllowScripts (){
  PRBool result;
  mPrototypeBinding->GetAllowScripts(&result);
  ...
  JSContext* cx = (JSContext*) context->GetNativeContext();

  nsCOMPtr<nsIDocument> ourDocument;
  mPrototypeBinding->XBLDocumentInfo()->GetDocument(getter_AddRefs(ourDocument));
  nsIPrincipal* principal = ourDocument->GetPrincipal();
  if (!principal) {
    return PR_FALSE;
  }

  PRBool canExecute;
  nsresult rv = mgr->CanExecuteScripts(cx, principal, &canExecute);
  return NS_SUCCEEDED(rv) && canExecute;
}
```

図 5：Seamonkey1.1.12 のソースコード(CVE-2008-5023)

本実験では上記ソフトウェアを Ubuntu8.04(x86-64)上の gcc-4.2.4 で，複製元と複製先ソフトウェアのソースコードを同じコンパイルオプションでコンパイルしたものを利用した．

また本実験では，コードクローンが存在していないと考えられる実行ファイルのデータセットとして，Ubuntu12.04(x86-64)の/bin と/usr/lib 配下に存在している実行ファイル 432 個を収集した．

実験では類似度算出の為のスコアを match =+2, mismatch = -2, opengap = -3, extendedgap = -0.5 とする．本実験結果を表 1 に示す．

表 1: 実験 1 の結果

CVE 番号	複製元	複製先	脆弱性箇所との類似度	修正済み脆弱性箇所との類似度	データセット中の最大類似度
CVE-2008-4316	Glib	Seahorse	60.7%	11.5%	9.2%
CVE-2008-5023	Firefox	Seamonkey	68.8%	38.0%	9.7%

実験の結果，CVE-2008-4316 のケースでは 60.7%の類似度が算出され，抽出された箇所も複製先脆弱性箇所に該当する部分だった．また，CVE-2008-5023 のケースでは 68.8%の類似度が算出され，抽出された箇所も複製先脆弱性箇所に該当する部分だった．またデータセット間での類似度算出においては，複製元脆弱性箇所に対し，最大でも 9.7%の類似度であり，コードクローンが存在した場合に比べ，十分に低い類似度算出結果になった．またこの事から閾値として適切な類似度は 20%であると考えられる．

本実験により提案手法を用いて算出された類似度から，複製先が一部改変済みのコードクローンの脆弱性でも特定可能であるという結果が得られた．

4.2 実験 2

実験 2 では，提案手法により既知の脆弱性を用いて実際に未発見のコードクローンの脆弱性が発見可能であるかを評価する実験を行った．具体的には，Microsoft 社の製品に存在した 14 個の脆弱性の脆弱性箇所用いて，収集した実行ファイル 40945 個に対して，本手法による検査を実施した．

実験に利用した脆弱性箇所と検査対象実行ファイルについて述べる．まず脆弱性箇所(未修正の脆弱性箇所と，修正済み脆弱性箇所)は，CVE 等を参考に脆弱性箇所となる関数を特定後 Microsoft 社が配布する修正パッチ(x86 版)を利用して収集した．収集した脆弱性一覧を表 2 に示す．

表 2: 対象脆弱性一覧

CVE 番号	脆弱性の種類	関数名	ファイル名
CVE-2015-1635	整数オーバーフロー	UlpParseRange	http.sys
CVE-2014-0301	メモリの二重解放	LoadJPEGImage NewBuffer	qedit.dll
CVE-2013-5058	整数オーバーフロー	RFONTOBJ:: bTextExtent	win32k.sys
CVE-2013-0030	バッファオーバーフロー	SavePathSeg	vgx.dll
CVE-2011-2005	メモリ管理の不備	AfdJoinLeaf	afd.sys
CVE-2011-0658	整数アンダーフロー	_PictLoad MetaFileRaw	oleaut32.dll
CVE-2010-0816	整数オーバーフロー	CPOP3Transport:: ResponseSTAT	inetcomm.dll
CVE-2010-0028	整数オーバーフロー	CBMPStream::Write	mspaint.exe
CVE-2008-4250	バッファオーバーフロー	sub_5925A26B	netapi32.dll
CVE-2008-4038	バッファアンダーフロー	SrvIssueQuery DirectoryRequest	srv.sys
CVE-2007-1794	整数アンダーフロー	CDownloadSink:: OnDataAvailable	vgx.dll
CVE-2007-0024	整数オーバーフロー	CVMLRecolorinfo:: InternalLoad	vgx.dll
CVE-2006-4691	バッファオーバーフロー	NetpManageI PCConnect	netapi32.dll
CVE-2006-0021	DoS	IGMPRcvQuery	tcpip.sys

次に検査対象となる実行ファイルとして，①VirusTotal に投稿されていた良性ソフトウェア(マルウェアとして検知されず，かつ NSRL[11]に登

録されている実行ファイル)と,②最新版(2015 年 4 月時点)各種 WindowsOS(x86)に存在している実行ファイルを収集した. 収集した実行ファイルの詳細について表 3 に示す.

表 3: 収集した実行ファイル一覧

収集元	ファイル数
Virus Total(NSRL)	7580
Windows XP	3479
Windows Vista	6933
Windows 7	5981
Windows 8.1	5048
Windows Server 2003	3984
Windows Server 2008	7940

本実験では, 収集した 14 個の既知の脆弱性箇所を用いて, 実行ファイル 40945 個に対して本手法を用いて検査を実施した. また本実験は実験 1 と同様のスコア設定を利用し, 類似度の閾値は 20%とした.

検査の結果発見された, 複製先の脆弱性と考えられる脆弱性候補一覧を表 4 に示す.

表 4: 発見された脆弱性候補一覧

CVE 番号	複製元	複製先	脆弱性箇所との類似度
CVE-2008-4250	netapi32.dll (5.1.2600.2952)	netlogon32.dll (5.2.3790.1830)	37.7%
CVE-2011-0658	oleaut32.dll (5.2.3790.4202)	olepro32.dll (6.1.7601.17514)	75.1%

発見された各脆弱性候補の詳細について述べる.

1）CVE-2008-4250 のケース

CVE-2008-4250 は, netapi32.dll 中に存在しているバッファオーバーフローの脆弱性で, Conficker が悪用する脆弱性として広く知られている. 実験の結果, VirusTotal 経由で収集された netlogon32.dll(5.2.3790.1830)中にも, 本脆弱性と同様の箇所を発見した. 発見した脆弱性箇所を調査した結果, 最新版の netlogon32.dll では脆弱性箇所が修正されていたものの, 複製元となった netapi32.dll より, 修正パッチが配布された時期が遅く, 過去に実際にコードクローンの脆弱性だったものであると判明した. 具体的には, netapi32.dll に対する修正パッチ(KB958644)が 2008 年 10 月に配布されたにも関わらず, netlogon32.dll に対する修正パッチ(KB961853)は 2009 年 1 月に配布されていた事が分かった.

本実験から, 本提案手法により既知の脆弱性箇所を用いて, 実際のソフトウェア中からコードクローンの脆弱性を発見する事が可能である事が示せた.

2）CVE-2011-0658 のケース

CVE-2011-0658 は oleaut32.dll に存在している, 整数アンダーフローの脆弱性である. 実験の結果, 各種 WindowsOS で収集された最新版の olepro32.dll(6.1.7601.17514:Windows7 版)中にも, 本脆弱性と同様の箇所を発見した.

しかし, olepro32.dll を調査した結果, 該当する関数は dll 中に存在はするものの, その呼び出し時において oleaut32.dll 中に存在する同名の関数に実行が転送される仕組みになっていた. そのため, 実際には悪用する事は難しいという結論に至った.

5 今後の課題とまとめ

本研究では, 実行ファイルを対象に, 類似文字列検索アルゴリズムを用いて複製された脆弱性の発見手法を提案し, 実験を通じて実際に複製先の脆弱性を発見する事が可能である事を示した. しかし, 発見された複製先脆弱性箇所の候補が実際に悪用可能であるか否かについては, 調査しない限り分からない. そのため今後は脆弱性個所の悪用可能性を判断出来る仕組みを検討していく. また本手法では複製元脆弱性箇所として, 脆弱性範囲が単一関数内に留まるもののみを対象としたが, 今後は脆弱性箇所が複数の関数に跨るような脆弱性への対応も今後の課題である.

参考文献

[1]A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitraş, “The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching,” in IEEE Symposium on Security and Privacy, San Jose, CA, 2015.
[2]Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, Tien N. Nguyen, “Detection of Recurring Software Vulnerability “, ASE'10, 2010.
[3]Jiyong Jang, Abeer Agrawal, and David Brumley,”ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions”, In Proceedings of the 33rd IEEE Symposium on Security and Privacy, 2012.
[4]Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, Heejo Lee, “A Scalable Approach for Vulnerability Discovery Based on Security Patches”, The 5th International Conference on Applications and Techniques for Information Security,Melbourne, Australia,November,2014.
[5]Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, Christian Rossow, “Leveraging semantic signatures for bug search in binary programs”, Annual Computer Security Applications Conference , New Orleans, USA, December 2014.
[6]Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, Thorsten Holz “Cross-Architecture Bug Search in Binary Executables”36th IEEE Symposium on Security and Privacy (Oakland), San Jose, May 2015.
[7]Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, “CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code,” IEEE Transactions on Software Engineering, vol. 28, no. 7, 2002, pp.654–670.
[8]Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu, “Deckard: Scalable and Accurate Tree-Based Detection of Code Clones,” Proceedings of the International Conference on Software Engineering, 2007.
[9]Needleman, S. B., & Wunsch, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology, 48, 443-453.1970.
[10]Gotoh Osamu, An improved algorithm for matching biological sequences. Journal of Molecular Biology. 162, 705-708, 1982.
[11] National Software Reference Library. http://www.nsrl.nist.gov/