以 Color Petri-Net 偵測 C 程式原始碼的緩衝區溢位

陳奕明	孫宇安	薛義誠
中央大學資訊管理系	中央大學資訊管理系	中央大學資訊管理系
中壢市五權里 2 鄰 38 號	中壢市五權里 2 鄰 38 號	中壢市五權里 2 鄰 38 號
cym@im.mgt.ncu.edu.tw	s8423042@cc.ncu.edu.tw	ycshiue@im.mgt.ncu.edu.tw

摘要

在電子商務日益流行而安全性又先天不足的網際網路環境中，軟體的安全問題日益受到重視，其中又以用 C 語言撰寫的軟體安全問題最嚴重。這是因為 C 語言的標準函式沒有自動檢查變數的宣告界限，容易造成緩衝區溢位(Buffer Overflow)問題。為解決緩衝區溢位的問題，本論文提供一個方便的檢驗方法來檢查 C 語言的原始程式碼是否有潛在的緩衝區溢位問題存在。我們採用具有正規理論基礎的 Color-Petri Net 方法，並使用 Design/CPN 工具來模擬 C 程式原始碼中字串處理的情形。和其他檢查原始碼緩衝區溢位問題的方法比較，我們的方法具有可處理程式動態流程(Dynamic Flow)的優點。論文中我們將介紹此方法的分析流程、方法，並以實例說明此方法的用法並且將結果和 Rats 方法比較，證明我們的方法在偵測精確度方面的確較後者為佳。

關鍵字：緩衝區溢位，Color-Petri Net，正規理論，軟體安全

1. 簡介

隨著電子商務時代的來臨，越來越多重要的系統與服務在網際網路上運作。由於 TCP/IP 協定先天上安全性設計不足，所以在不安全的網路環境中，軟體的安全性顯得格外重要。其中又以長久以來採用 C 語言撰寫的軟體安全性問題最為重要，這是由於 C 語言並沒有自動檢查變數宣告界限，使得許多呼叫 C 語言標準函式庫的程式面臨緩衝區溢位的問題[2]。

緩衝區溢位的問題主要來自於撰寫程式時，使用字串函數不夠謹慎，沒有做好長度確認，導致攻擊者可以針對緩衝區溢位的漏洞，把過量的資料寫入緩衝區中，覆蓋緩衝區中原本的資料。藉由這樣的方式，攻擊者可以把一些惡意的程式碼注入一個執行緒中，篡改程式的執行流程，進一步奪取程式的控制權。根據美國 CERT/CC 公佈的-截至 2001 年五月底為止，網路掃描漏洞列表[3]，發現在共計 37 個常見的掃描漏洞中，有 22 個與緩衝區溢位問題相關，比例高達 59% ，由此可見緩衝區溢位問題的嚴重程度。

目前對於緩衝區溢位問題及其他的軟體安全性問題的處理方法為「滲透與修補」[4]，亦即在發現軟體有漏洞之後，由系統廠商提供修補程式，讓使用者自行安裝修補程式以防堵軟體的安全性漏洞。雖然這種方法已經行之有年，但是治本的做法應該是在軟體商業化之前，就積極地盡量減少安全性的漏洞，而非消極地採用亡羊補牢的事後補救辦法。

因此本文的目的在於對已經寫好的 C 語言程式碼，提供一方便的檢驗方法檢查是否有潛在的緩衝區溢位問題，換句話說，對於仍然採用 C 語言開發軟體的程式設計師而言，希望提供他們有效的稽核程式碼方法，在軟體商業化或是系統上線之前，修正潛在的緩衝區溢位問題，減低軟體遭遇安全威脅的可能性。

我們的方法是在假設可以取得軟體原始

碼的情況下，將原始碼轉換成 Color Petri Net Graph[5]，然後利用 Petri-Net[6]固有的正規化分析(Formal Analysis)方法預測是否有潛在的緩衝區溢位問題。雖然我們處理緩衝區溢位的方法只屬於靜態分析(Static Analysis)，而不是在程式執行時去偵測，但由於我們採用了 CPN 分析方法，所以我們可以利用 CPN 的分析工具-Design/CPN[7]，來分析程式碼的動態流程。論文中我們會以實例來說明此分析法的流程，並實際分析 Wu-ftpd 等實際網路程式，然後和 Rats 工具[8]比較，證明在精確度方面優於 Rats 工具，且的確可偵測緩衝區溢位問題。

本論文分為五節。第二節將介紹緩衝區溢位問題靜態分析方法；第三節將介紹以 Color Petri Net 理論為基礎，偵測緩衝區溢位問題的方法與模型；第四節將對於所提出的方法及模型做實例探討；第五節為結論以及未來可繼續研究的方向。

2. 相關研究

現有的緩衝區溢位防禦方法基本上可以分為三類：第一類的方法主要是以撰寫安全的程式碼為主，例如[9]，目的在處理字串時避免字串的溢出。第二類的方法是從保護返回位址的角度防禦，例如[10]；在中央處理器要讀取返回位址時，先檢查返回位址是否有被修改過，如果有而且和原來保留的返回位址備份不相同，就中止程式的執行。第三類的防禦方法則是禁止攻擊程式在堆疊中執行[11]，也就是說，儘管緩衝區溢出置換了返回位址，新的返回位址也正確地指向攻擊程式碼所在的記憶體位址，但是只要攻擊程式碼是存放在堆疊中，而把堆疊設定為不可執行的狀態，那麼緩衝區溢位攻擊就不能成功。

以上方法各有其設計上的著眼點，但考量到在程式開發的過程中，越早修正錯誤造成的影響越小[12]，所以我們針對第一類的方法，亦即靜態分析程式碼的方法，來作改進。

靜態分析程式碼方法可以歸納為以下四種（參見表 1）：(1)過濾危險函數呼叫, (2)以限制式為基礎, (3)分析程式流程以及(4)其他方法，分別介紹如下：

方法 1. 過濾危險函數呼叫

在過濾危險函數呼叫方面，以表 1 中之 ITS4[9]、Rats[8]方法為代表。ITS4 是 C 語言及 C＋＋語言原始碼的靜態漏動掃瞄工具，其原理主要是將原始碼與漏洞資料庫做比對，在原始碼中過濾出不安全的函數呼叫，如 strcpy、strcat 等等。與 grep 不同之處在於，ITS4 處理的是未經過編譯器預處理的原始碼。另外，除了緩衝區溢位漏洞之外，ITS4 也針對軟體安全上的其他方面做檢測，例如競速（race

表 1：靜態分析程式碼相關研究分類表

	Flow Insensitive	Locate statement	Pointer Handling	Theory Based-On	False Positive	False Negative	Overhead
ITS4、Rats	Yes	Yes	No	Vulnerable Database	Very High (4/79)	Close to 0	Low
Wanger	Yes	No	No	Set Constraint	High (4/44)	Low	Low
Symbolic Bounds Checking	No		Yes	Symbolic Constraint	Low	Low	High
Safe C	No	N/A	Yes	Safe Pointer Transformation	N/A	N/A	High
WISC Debugging tool	No	Yes	Yes	Add tags	Not mentioned	Low (None)	High
Our Model	Yes	Yes	Yes	Color Petri Net	Low	Low	High

condition）問題。

由於 ITS4 並不分析程式流程，也不比對字串長度，只單純做函數的比對，因此在偵測緩衝區溢位問題方面，誤報率非常高（75/79）[9]。不過在效能方面的表現佳，在 Pentium 等級的機器上每秒可以過濾 7000~9000 行左右的原始程式碼，適合分析大型軟體。

方法 2 以限制式為基礎

以限制式為基礎(Constrain-based)靜態偵測緩衝區溢位問題的方法，以表 1 中之 Wanger 方法[13]為代表。此方法主要分為兩大階段：Constraint Generation 及 Constraint Solver，第一階段（Constraint Generation）目的在依據 C 語言程式碼產生對應的限制式，第二階段（Constraint Solver）則是處理所產生的限制式，以偵測是否有溢位的情況產生。

其原理為對於程式碼中每一個字串變數 s，定義兩個屬性，分別為 len(s)表示字串目前的長度，及 alloc(s)表示字串宣告的長度。在第一階段中針對 C 語言程式碼，每一字串變數 s 皆會產生對應的 len(s)限制式及 alloc(s)限制式，接著針對每一個字串變數 s，產生 len(s)的對應圖，並在圖中尋找 len(s)的延伸路徑（augmenting path），可求出 len(s)的上下限，以同樣的方法求得 alloc(s)的上下限後，比對 len(s)與 alloc(s)的上下限，若符合安全條件 len(s) ≤ alloc(s)就表示沒有緩衝區溢位的問題，若否，則變數 s 有潛在緩衝區溢位的問題。

以限制式為基礎的方法和 ITS4 一樣不分析程式流程，因此在誤報率方面仍不能令人滿意，大約有 90% （40/44）的誤報率。但是和 ITS4 不同的地方在於長度比對，限制式方法有處理長度，因此雖然誤報率有 90% [13]，仍然是相對來說比較精確的偵測方法。

方法 3.分析程式流程

以表 1 中之 Symbolic Bounds Checking [14]方法為主。此分析程式流程的方法是在原始碼中，針對指位器(pointer)、陣列索引及使用的記憶體區塊，各設有一「上限」及「下限」，然後以上下限的比對以判斷是否有溢位問題存在。基本上，此方法主要仍是以限制式為基礎，但是經過特殊演算法處理後，可以將原始碼的複雜度減低到線性。此方法的優點是並不只可以分析緩衝區溢位問題，也可以處理競速等軟體安全性的問題。

方法 4.　其他

除了上述三種主要的靜態分析程式碼方法之外，尚有其他如 Safe C[15]、WISC Debugging Tool[16]等靜態分析程式碼方式。Safe C 及 WISC Debugging Tool 皆以處理指位器相關問題為主，Safe C 的做法是將原始程式碼中，與指位器相關的部分轉換成安全指位器的型態。而 WISC Debugging Tool 則是採用在原始碼中加入標籤的方法。

這兩種方法的主要缺點都是 Overhead 過重，這也是許多試圖處理指位器方法的常見問題。但是同樣地，處理指位器相關問題的方法，皆有精確度提高、誤報率低的優點。

3. Color Petri Net 與 C 程式原始碼緩衝區溢位分析

3-1. Color Petri Net 簡介

Color Petri-Net(CPN)是 1992 年由 Kurt Jesen [5]所提出，和 Petri Net 不同的地方是他增加了 Color Set（彩色集）的觀念，Color Set 相當於資料型別(Data Types)，CPN 的正式定義如下：

CPN＝（∑, P, T, A, N, C, G, E, I），其中

∑為一組有限個數的型別(Types), 又稱為彩色集（color sets）.

P 為一組有限個數的 places.

T 為一組有限個數的 transitions.

A 為一組有限個數的 arcs，其中：

$P \cap T = P \cap A = T \cap A = \varnothing$

N 為節點函數(Node Function).

N: A-> P×T∪T×P.

C 為彩色函數(color function). C: P -> ∑.

G 為導引函數(guard function).

G: T -> expressions， 而且：

∀t∈T：[Type（G（t））＝B ∧ Type（Var（E（a）））⊆∑]

其中 p 是 N（a）的 place。

E 為連結表示式函數(arc expression function).

E: A -> expressions ， 而且：

∀a∈A：[Type（E（a））＝C（p）MS ∧ Type（Var（E（a）））⊆∑]

I 為初始函數(initialization function.

I: P ->closed expressions， 而且：

∀p∈P：[Type（E（a））＝C（p）MS].

由於我們在分析原始程式碼的過程中，需要分析變數的宣告長度、目前長度及相關指位器等屬性，因此採用具有資料型別的 CPN 有助於進行緩衝區溢位分析且可以利用 Petri-Net 原有的正規化分析優點。

3-2 原始程式碼轉換成CPN圖形的處理原則

我們採用 CPN 方法將程式碼轉換成 CPN 圖形之後，就可以使用 Petri-Net 的正規分析模擬方法，以分析該 CPN 圖形是否會進入「Dangerous」狀態，並且藉由是否會進入「Dangerous」狀態判斷是否有緩衝區溢位問題存在。

上述過程看似簡單，但實際上在將原始程式碼轉換成 CPN 圖形時，有許多細節問題需要考量。以下我們先介紹在採用靜態分析程式碼方法以偵測是否緩衝區溢位問題中，常見的處理方法及常遭遇的困難。首先以圖 1 為例，緩衝區溢位問題存在於程式碼的第三行 strcpy(buffer, str)。而會發生緩衝區溢位問題的原因為，根據圖 1 的第二行程式碼，變數 buffer 宣告時的長度為 16，但是變數 str 的長度為 256。有許多緩衝區溢位問題的防禦方法是判斷字串長度，因此我們在將程式碼轉換成 CPN 圖形中，必須注意字串變數的兩大屬性：「宣告長度」與「目前長度」。

```
1   void function(char *str) {
2           char buffer[16];
3           strcpy(buffer, str);
4                       }
5   void main() {
6       char large_string[256];
7       int i;
8       for( i = 0; i < 255; i++)
9           large_string[i] = 'A';
10          function(large_string);
11                  }
```

圖 1：緩衝區溢位原始碼範例一（資料來源： [17]）

```
1   void simple() {
2       char s[20]
3       char *p
4       char t[10]
5           strcpy(s, "hello");
6           p = s+5;
7           strcpy(p," world!!");
8           strcpy(t,s);
9           }
```

圖 2：緩衝區溢位原始碼範例二（資料來源： [13]）

```
1    void simple() {
2        char s[20]
3        char t[10]
4        strcpy(s, "hello");
5        strcpy(t,s);
6            }
```

圖 3：忽略指位器之緩衝區溢位範例原始碼

另外一個重要的問題是指位器，程式設計師經常使用指位器處理字串的複製、截斷、移動等等，如果在轉換程式碼時忽略了指位器，則會大幅降低偵測的精確度。例如在圖 2 的例子中，在第六行與第七行程式碼中，使用指位器 p 複製字串，改變了字串 s 的長度，才會在第八行程式碼產生緩衝區溢位的問題。在圖 2 中，如果忽略與指位器有關的程式碼部分，則程式碼變成下面的圖 3。在圖 3 中，第四行與第五行並沒有超出宣告長度的問題，偵測不出在第五行程式碼中有緩衝區溢位的問題，但是在圖 2 中的同一行程式碼卻是確實存在緩衝區溢位的問題。由此可見在轉換程式碼成為 CPN 圖形的過程中，必須處理指位器的相關問題。

綜合以上所述，在本研究方法中，歸納出程式碼與 CPN 圖形間的轉換原則，條列如下：

1. 簡化程式碼：只處理與字串變數及指位器有關的程式碼
2. 每一個字串變數皆具備三個屬性（1）variable_length（2）variable_allocated（3）related_pointer
   variable_length 表示字串目前的長度，variable_allocated 表示字串宣告時的長度，而 related_pointer 為指向該字串的指位器。
3. 每一個指位器變數皆具備三個屬性（1）pointer_length（2）parent_string（3）related_position
   pointer_length 表示從該指位器起算，字串目前的長度，related_position 表示字串與指位器的相對位置，parent_string 表示目前指位器指向的字串變數。
4. 每一個函數呼叫對應為一個 Transition
5. 函數呼叫中每一個傳遞的參數對應為一個 Place
6. 程式碼中若使用到有危險性的函數呼叫，則在轉換之前加上長度判斷的步驟，若無法通過長度判斷，就會進入「Dangerous」狀態

詳細的節點轉換，以及字串與指位器處理等規則，請參見【1】

4. 實例分析

在本節第一小節中，我們以圖 1 的原始程式碼為範例，說明原始程式碼之轉換及分析過程。接著在第二小節中，以 wu-ftpd 2.4 版中部分原始程式碼為實例說明本方法在偵測緩衝區溢位問題方面所得之結果。

4-1 範例原始程式碼轉換

本方法的分析過程分為三階段。第一階段為將原始程式依節點轉換規則轉成 CPN 節點並建成 CPN 圖形。以圖 1 的程式碼為例，使用 CPN 圖形分析工具 Design/CPN 將該程式碼以圖形化的方式呈現如圖 4 所示。有了 CPN 的基本圖後，接著進行第二階段 Arc 參數與限制式之轉換，結果如圖 5 所示，注意圖 5 中的 Arc 較圖 4 的 Arc 多了 Color-set 的表示式。

經過第二階段轉換之後，所有的 Arc 皆已設定好參數及限制式，最後進入第三階段，也就是處理字串與指位器相關問題，得到最後轉換結果如圖 6 所示。在圖 6 中以橢圓形虛線標示的連接 Place p 與 Transition strcpy 之 Arc，是字串與指位器處理的結果，而 Transition strcpy 的兩個 Arc_out 的參數及限制式也相對修正。此時判斷是否進入 Dangerous 狀態判斷式中，不再是以單純的 Place s 的目前長度為基準，而是以函數 update 回傳值為判斷基準。

本範例模擬結果如圖 7 所示，會在最後一個 strcpy Transition 後進入 Dangerous 狀態，在圖中以橢圓形虛線標示。（“13, 10”）表示目前長度為 13 的字串溢出宣告之前長度為 10 的字串。

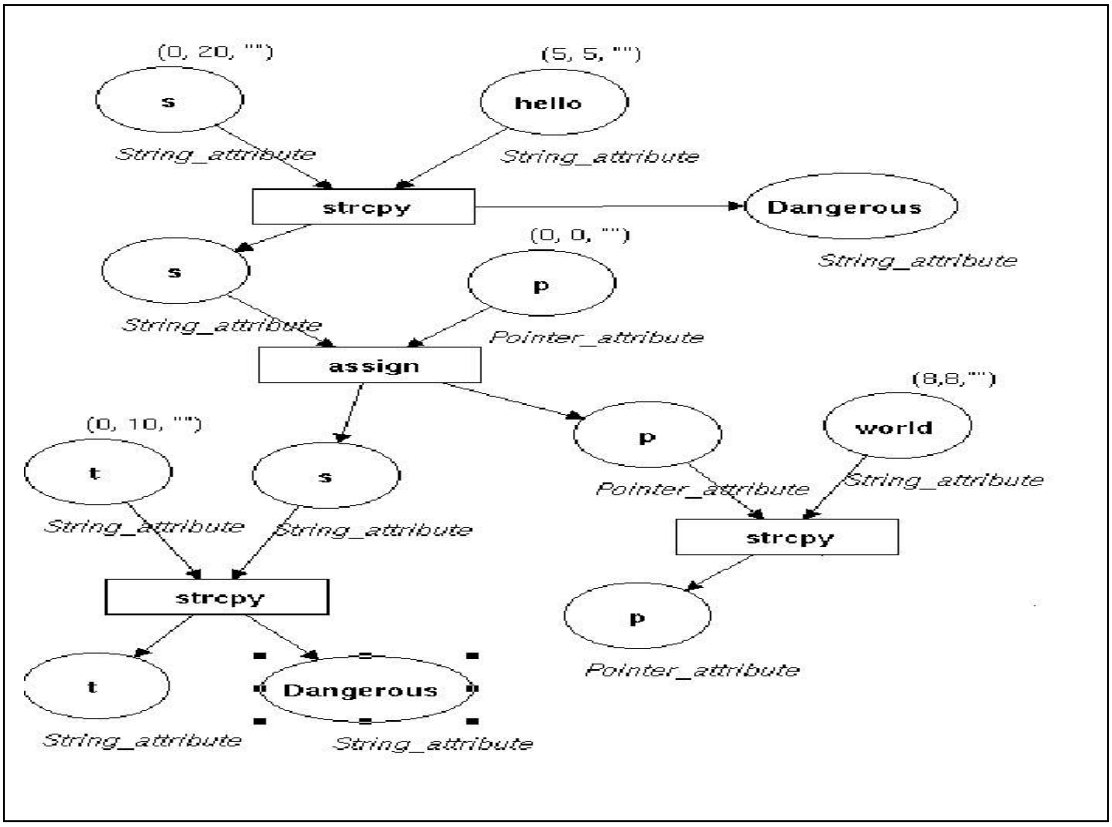

圖 4：圖 1 範例程式之第一階段轉換結果圖

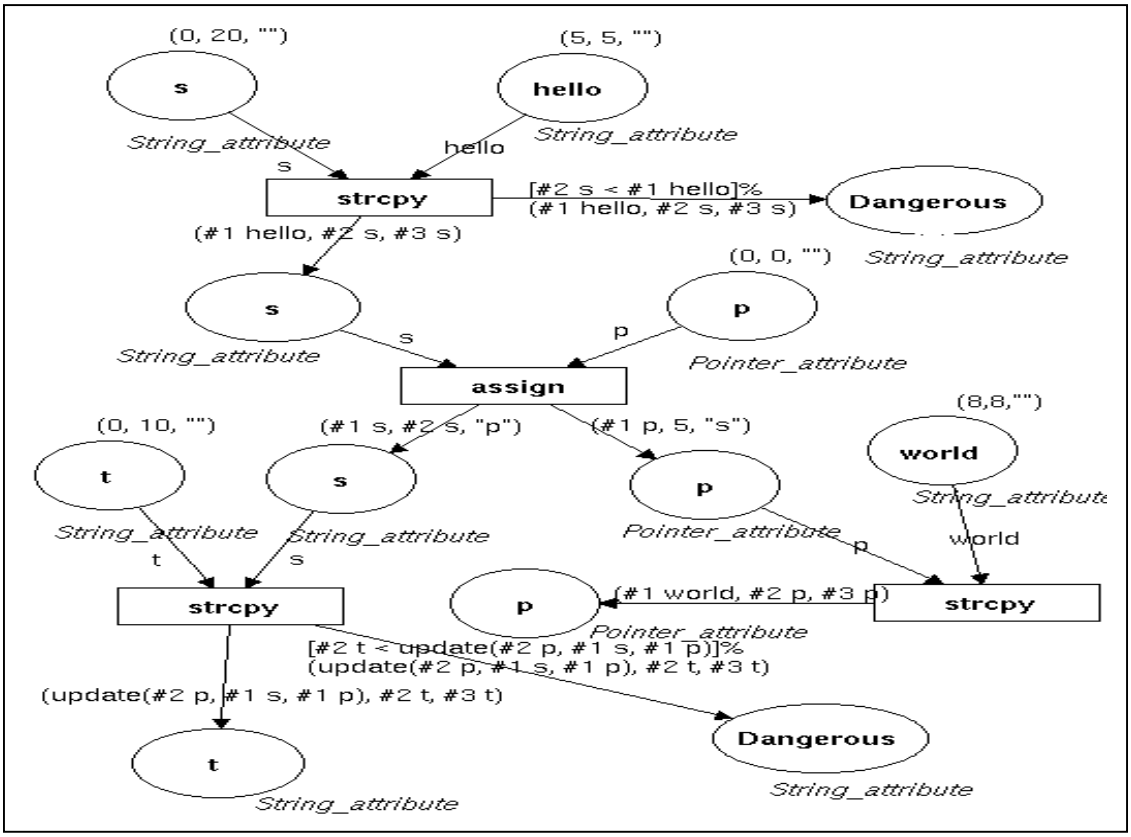

圖 5：圖 1 範例程式之第二階段轉換結果圖

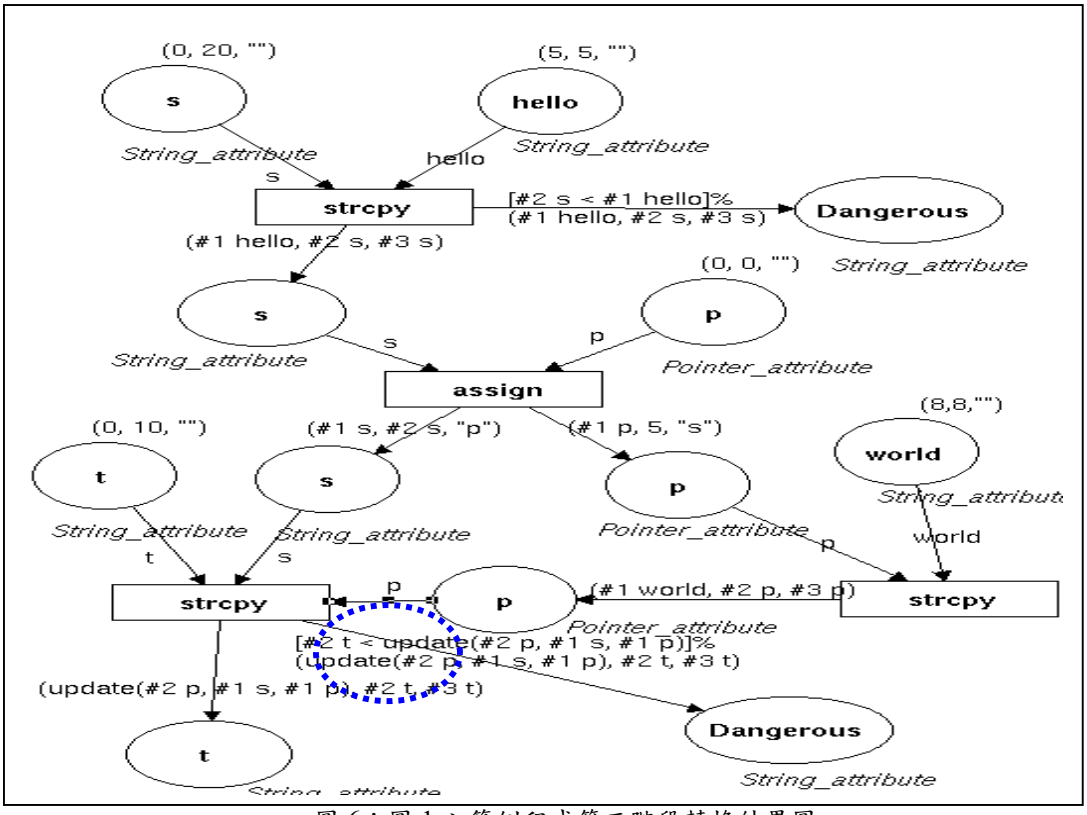

圖 6：圖 1 之範例程式第三階段轉換結果圖

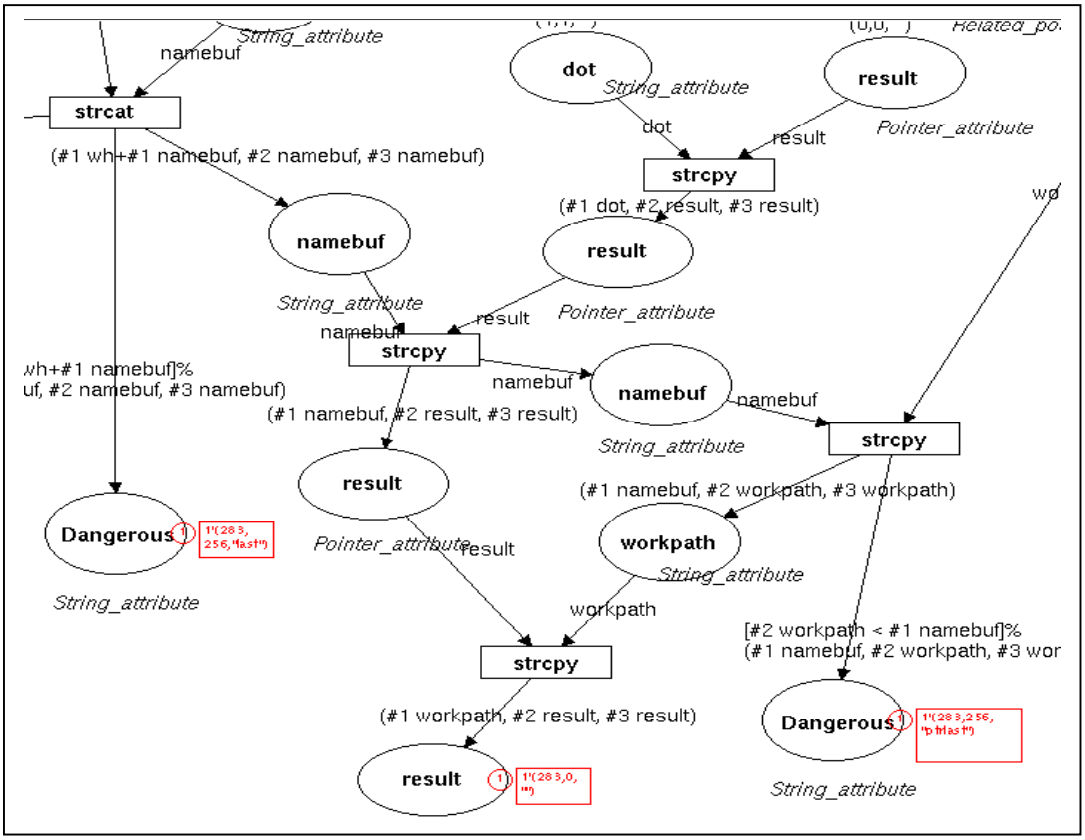

圖 7：圖 1 之 CPN 圖形模擬結果圖

4-2 WU-FTPD 原始碼轉換實例探討

Wuarchive-ftpd[18], 通常簡稱為 wu-ftpd，是在網路上最常見的檔案存取伺服器軟體。但是 wu-ftpd 發展到現在，各個版本都被發現有安全上的漏洞。從 1993 年四月到 2001 年一月，共計有十個與 wu-ftpd 相關的安全通報，因此我們選擇 wu-ftpd 做偵測緩衝區溢位問題方面的實例分析。

我們分析的版本是 wu-ftpd 2.4，這個版本中的 realpath.c 檔案在 CERT 的安全通報中列為有漏洞的原始碼。wu-ftpd 2.4 中的 realpath.c 提供將使用者輸入的路徑轉換伺服器上的對應路徑之功能，因此會有許多與字串處理有關的程式碼。該程式碼是 wu-ftpd 發展團隊為了置換原本 C 語言函式庫中不安全的 realpath()函數而撰寫的，未料在撰寫的過程中仍有字串長度判斷上的疏漏。因此我們將這個檔案轉換成 CPN 圖形，試圖偵測其中的緩衝區溢位問題。

realpath.c 原始程式碼共計 161 行，我們將其中與字串處理有關的原始程式碼進行三階段轉換，然後進行模擬。模擬結果如圖 8 所示，顯示當輸入字串長度與工作路徑長度之和超過 256 時，會分別經過一 strcat 的 Transition 及 strcpy 的 Transition 之後進入 Dangerous 狀態。與原始碼比對之後，發現是在第 122 行的 strcat(namebuf, where) 及第 157 行的 strcpy(workpath, namebuf)。而採用根據 ITS4 方法開發之 Rats 工具[Rats 01]，偵測 realpath.c 緩衝區溢位漏洞之結果如圖 9 所示。在圖 9 中以橢圓形標示的部分，顯示出 Rats 也有警告第 122 行及第 157 行可能有潛在的緩衝區溢位問題。但是 Rats 針對 realpath.c 檔案一共有十一個警示點，也就是說誤報率高達 9/11（約 82% ），顯然偏高。

5. 結論

本論文之主要貢獻在於整理歸納出緩衝區溢位的靜態分析防禦方法、指出目前相關研究的缺點，並發展出一套完整的程式碼轉換至 Color Petri Net 的規則來改善這些缺點。我們的方法具有一般靜態分析法所缺少的可處理動態流程等優點。我們使用 Design/CPN 工具來模擬我們轉換後的 CPN 圖形。經由 Wu-ftpd 等實際網路程式的分析，和 Rats 工具比較，我們的方法證明在精確度方面優於 Rats 工具，且的確可達到偵測緩衝區溢位問題的功能。

目前本方法的限制是在將原始碼轉換成 CPN 圖形的過程中，由於我們只處理字串長度相關的屬性，並不記錄字串內容。因此當 if-then-else 的條件判斷式是和字串內容比對有關時，我們的處理方法是假設兩個路徑都會經過，字串長度則是取兩者的最大值。這樣處理的缺點是會增加誤報率，但在不記錄字串內容的情況下，目前尚無找到更好的處理方法。

此外，本方法處理的是在記憶體堆疊區發生的緩衝區溢位問題，未來若在指位器部分加上處理指位器與資料結構間的相對情況，則可以進一步處理在動態配置記憶體區（heap）發生的溢位問題。

最後，在我們分析 wu-ftpd 程式原始碼時，發現在部分原始碼中有執行期才能得知長度的字串變數，目前我們是採用亂數產生的方式模擬此類字串長度的改變。因此，未來研究方向或許可與 Fault Injection 方法結合[19]來處理執行期才能得知長度的字串變數問題。

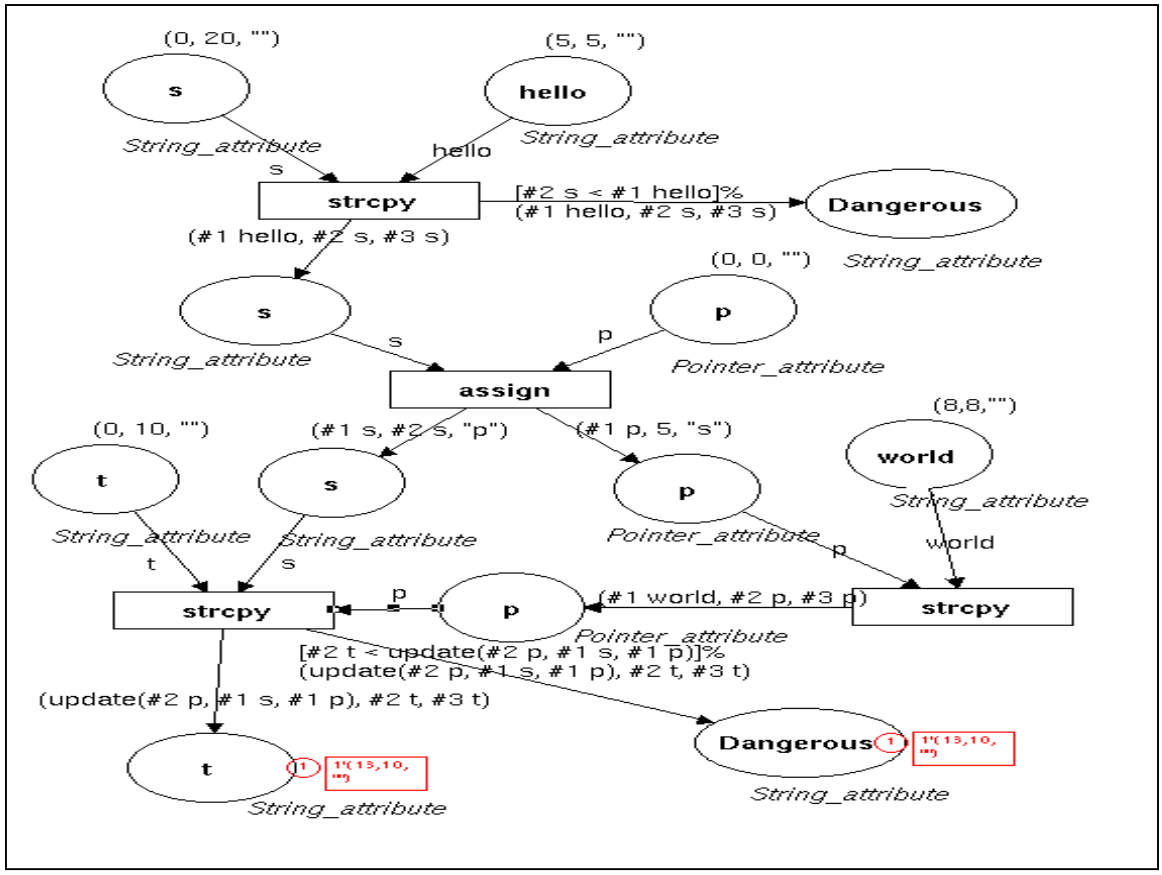

圖 8：realpath.c 模擬結果圖

```
[root@Meredith rats-0.9]# ./rats realpath.c
realpath.c: 45: High: realpath
Be sure the destination buffer is at least MAXPATHLEN big.  This function may still internally overflow a static buff
er, try to avoid using it.  If you must, check the size the path your pass in is no longer than MAXPATHLEN

realpath.c: 48: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely.  They are prime targets for buffer overflow
attacks.

realpath.c: 57: High: strcpy
realpath.c: 117: High: strcpy
realpath.c: 126: High: strcpy
realpath.c: 134: High: strcpy
realpath.c: 145: High: strcpy
realpath.c: 150: High: strcpy
realpath.c: 154: High: strcpy
realpath.c: 157: High: strcpy
realpath.c: 159: High: strcpy
Check to be sure that argument 2 passed to this function call will not more
data than can be handled, resulting in a buffer overflow.

realpath.c: 122: High: strcat
realpath.c: 143: High: strcat
Check to be sure that argument 2 passed to this function call will not more
data than can be handled, resulting in a buffer overflow.
```

圖 9：採用 Rats 工具偵測 realpath.c 緩衝區溢位問題結果圖

參考文獻

【1】孫宇安，採用 Color Petri-Net 方法偵測程式原始碼緩衝區溢位問題，國立中央大學資訊管理研究所碩士論文，民國 90 年 6 月

【2】Arash Baratloo, Timothy Tsai and Navjot Singh. “Libsafe ： Protecting Critical Elements of Stacks.” Bell Labs, Lucent Technologies, December 1999.

【3】Cert 2001 Vulnerability Summary Report

【4】Gary McGraw. “Testing for Security During Development: Why we should scrap penetrate-and-patch.” *IEEE Aerospace and Electronic Systems,* April 1998.

【5】K. Jensen. “Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol 1 ： Basic Concepts,” 1992. *Monographs in Theoretical Computer Science, Spring-Verlag.*

【6】James L. Peterson. “Petri Net Theory and the Modeling of Systems.” *Prentice-Hall, N.J.,* 1981

【7】Design/CPN Web site http://www.daimi.au.dk/designCPN/

【8】http://www.securesw.com/rats/

【9】John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. “ITS4 : A Static Vulnerability Scanner for C and C++ Code.” In *Proceedings of the 16th Annual Computer Security Applications Conference. New Orleans, Louisiana,* December 2000.

【10】Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle and Erik Walthinsen. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.” In *Proceedings in the 7th USENIX Security Symposium*, January 1998

【11】”Solar Designer”. Non-Executable User Stack. http://www.openwall.com/linux/

【12】Roger S. Pressman. “Software Engineering：A Practitioner’s Aprroach, Fourth Edition.” *McGraw-Hill,* 1997.

【13】D. Wagner, J. Foster, E. Brewer, and A. Aiken. “A first step towards automated detection of buffer overrun vulnerabilities.” In *Network and Distributed System Security Symposium, San Diego, CA,* February 2000.

【14】R. Rugina and M. Rinard. “Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions.” *SIGPLAN Conference on Programming Language Design and Implementation. Vancouver B.C., Canada,* June 2000.

【15】Todd M. Austin, Scott E. Breach and Gurindar S. Sohi. “Efficient Detection of All Pointer and Array Access Errors.” PLDI’94, ACM.

【16】Alexey Loginov, Suan Hsi Yong, Susan Horwitz and Thomas Reps. ”Debugging via run-time type checking.” In *Proceedings of FASE 2001: Fundamental Approaches* to Software Engineering, Genoa, Italy, April 2001.

【17】Nathan P. Smith. “Stack Smashing vulnerabilities in the UNIX Operating System.” http://millcomm.com/nate/machines/security/stack-smashing/nate/buffer.ps 1997.

【18】http://www.wuftpd.org