Code Breakers Journal

© The CodeBreakers-Journal, Vol. 1, No. 1 (2004)
http://www.CodeBreakers-Journal.com

Protecting Applications with Petri Nets

Robert Airapetyan, Polytechnical University of Odessa
and
Thorsten Schneider, University of Bielefeld

Abstract

Reverse Code Engineering of Software-Applications is often used by software pirates and crackers to extract code segments of compiled applications or to circumvent software protections. Preventing attacks like Bruteforce Attacks is a primal problem of software-protectionists. This paper illustrates the effectiveness of a concept using Petri nets to prevent software-piracy.

Keywords: *Petri Nets; Software Protection; Reverse Code Engineering; Dicotyledonous Oriented Multi-graphs*

Contents

I	**Introduction**	2
II	**Example of Petri nets usage**	3
III	**Method**	3
IV	**Difficulty of the Reachability Problem**	5
V	**Discussion**	6
VI	**Conclusions and Future Work**	6
	References	6
VII	**Attachment**	7

I. Introduction

Petri Nets are dicotyledonous oriented multigraphs [8]. They consists of four basic elements: ensemble of places P (schematically marked as circles), ensemble of transitions T (marked as lines), input function I and output function O. Input and output functions are linked with places and transitions. The input function I maps a transition t_j to ensemble places $I(t_j)$, which are called *input places* of the transition. The output function O maps the transition t_j to ensemble places $O(t_j)$, which are called *output places* of the transition. Oriented arks (arrows) connect places (nodes) with transitions, whereby some arks are moved from places to transitions and others vice versa.

Marking μ is an awardation of tokens to places of Petri net. A token is a primitive concept of Petri net (like place and transition). Tokens are used for defining the execution of a Petri net whereas the execution of the Petri net is under control of the defined tokens. Tokens are placed in circles (places) and control the transition execution in the net. The Petri net is executed by executing the transitions. Any transition can be executed only if it is allowed.

A transition is allowed, if each of its input places has an amount of tokens at least once equally to amount of arks from this place to the transition. For example if places p_1 and p_2 are inputs for t_4, then t_4 is allowed, if p_1 and p_2 has at least one token in each (see figure 1).

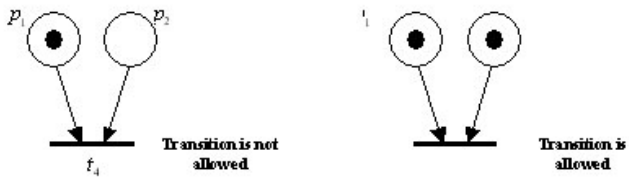

Fig. 1. Example of allowance of transitions within a Petri net.

For a transition t_1 with an input set $\{p_6, p_6, p_6\}$ the place p_6 must have at least three tokens within for allowing a transition t_1 (see figure 2).

The transition is executed by deleting all permitted tokens from their input places and follows by adding tokens to the output places - one token for each ark (see figures 3 to 6).

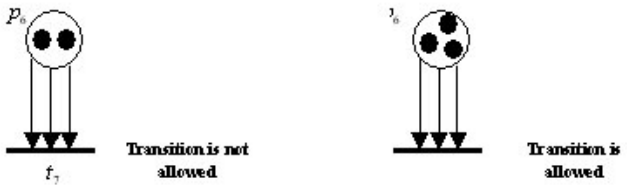

Fig. 2. Example of allowance of transitions within a Petri net for 3 tokens.

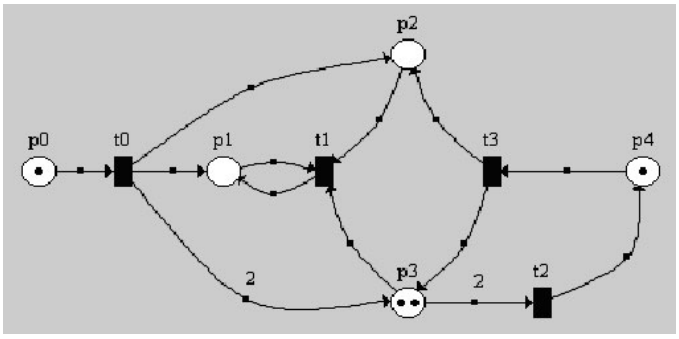

Fig. 3. In this net transitions t_0, t_2 and t_3 are allowed.

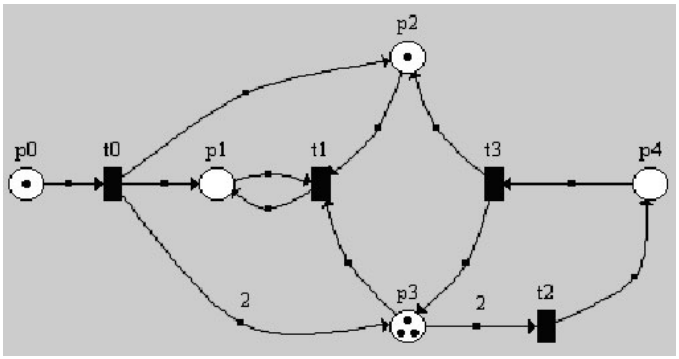

Fig. 4. Marking, which appears after execution of t_3.

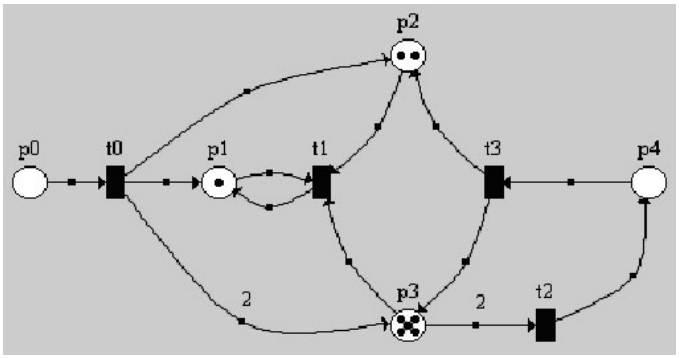

Fig. 5. Marking, which appears after execution of t_0. There are TWO arks from t_0 to p_3, that's why TWO tokens were added to p_3.

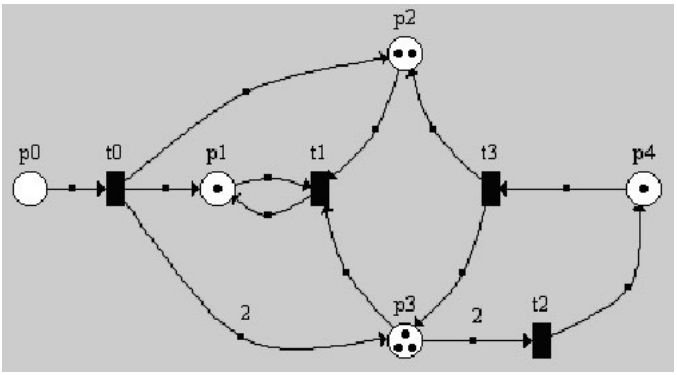

Fig. 6. Marking, which appears after execution of t_0. There are TWO arks from t_0 to p_3, that's why TWO tokens were added to p_3.

We can continue until there is at least one allowed transition. Finding of dead ends within Petri nets are described by other authors [7] and is not focused here.

II. Example of Petri nets usage

Dijkstra [1] offered in the year 1968 one of the most known problems - *The Five Chinese Sages Problem*. He describes the problem as it follows:

> *Five Chinese sages are sitting at the circle table and have a dinner. Between of each two sages is only one stick. But for eating each of them needs two sticks in a moment. Obviously, if all sages takes sticks from the left side and waiting sticks from right side they all will die through starvation (dead loop).*

Here, Petri nets can save their lives! Figure (7) describes the problem as Petri net. Places $P1...P5$ introduces sticks and all sticks are on the table at first moment (each place have a marker inside). Transitions T_i and E_i introduces sages states: $T_i - sage_i$ thinks, $E_i - sage_i$ eats. To pass from M_i state (obviously, no one can satisfy his hunger through his thoughts) to E_i state both sticks (on left and right sides) must be on the table at one moment.

III. Method

We use Petri nets to describe a theoretical appraoch in software protection schemes. Instead of using external software packages like *Petri Net Kernel* [2] we use a theoretical approach. Petri nets are an ideal tool for transitions modeling. It's not a secret, that most of protection

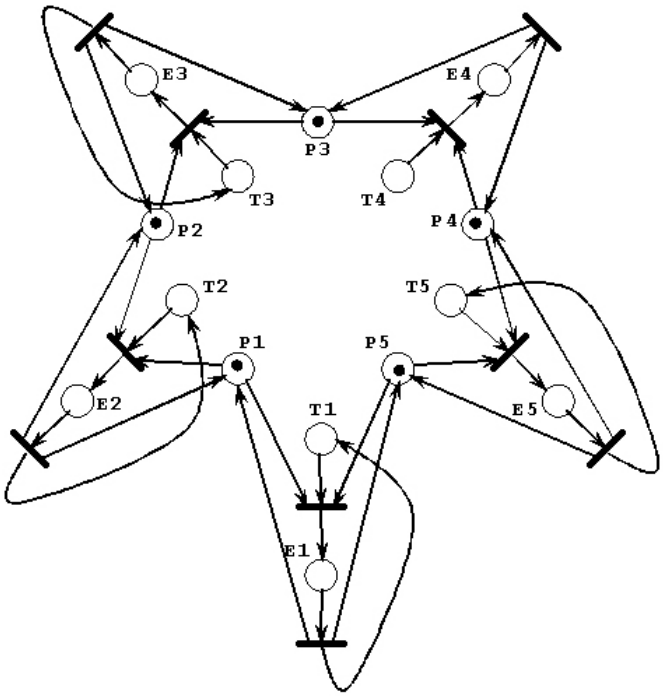

Fig. 7. Five Chinese Sages problem described by Dijkstra [1] as Petri net.

schemes reduce to hide notorious conditional jumps from long software cracker hands. Simplified this describes the *"very bad guy/good guy jump"*. Herewith coverage of a protection may consists of refined anti-debugging and anti-disassembling tricks and any other senseless thing which can be imagined but in the middle of that cover is just one decisive jump. We consider a simple example for representing graphically the introduced protection scheme.

We create a simple Petri net shown in figure (8). Imaging that on the stage of initial marking we can mark only places $p_0...p_3$. It is noticeable that place p_7 already has a token. Other places are inaccessible for marking. Then we assign the lowest priority for the t_2 transition - which means, that if both transitions t_2 and t_1 are allowed then transition t_1 will be executed - after which it (t_2 transition) will be executed if during initial marking places $p_0...p_3$ are marked in the manner $p_0 = 1, p_1 = 0, p_2 = 1, p_3 = 1$. In all other cases of initial marking transition t_2 will not executes.

Now we imagine that each transition is a thread which checks set of input bits (input places) and depending on result sets output bits (output places). So, thread t_3

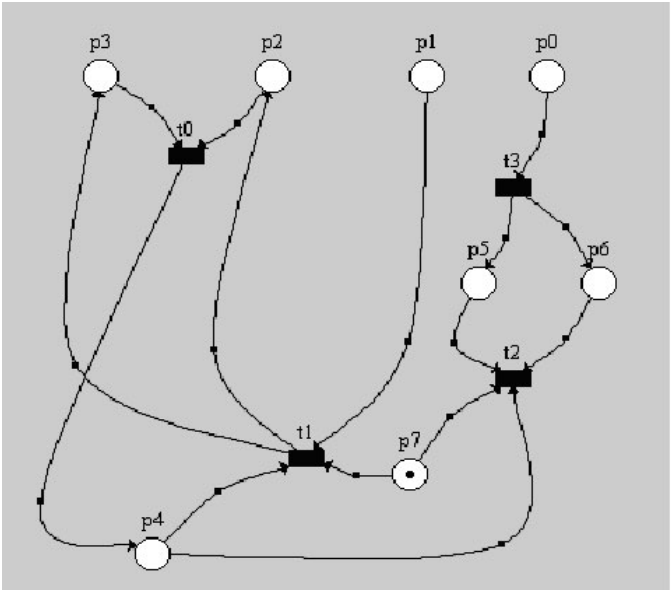

Fig. 8. A simple Petri net for a software protection task.

checks tokens (bits) in place p_0 and depending on result sets bits on output places (p_5 and p_6).The user has to register the application by inputing 4 bits of a key. The bits of the key (tokens) are placed to the places p_0, p_1, p_2 and p_3. It will be the *initial marking* of the net.

The application becomes registered if transition t_2 was executed. As we described the previously transition t_2 will be executed only with the initial marking $p_0 = 1$, $p_1 = 0$, $p_2 = 1$, $p_3 = 1$.

The problem of reachability is the main problem which exists when solving Petri nets (see section [3] (IV)). Many other problems are reduceable (in particular - problem of activity) to a reachability problem. When we try to decide problems of reachability using reachability trees practically we resort to the brute force method. In general we can reduce any problem to brute forcing. A software protection is defined by time and resources which we have to consume to break the protection in general. In our example we can easily find a correct key (right initial marking) with only $2^4 = 16$ tries. We can do this manually or heuristically within a few seconds. But we can easily increase our net automatically and hereunder complicating its structure and increase the key length. This is a very important characteristic of the net. Under increasing we understand the following: for each place which we can mark during initial marking we link-up a new net. Herewith those places lose their marking characteristics - this means they can not be marked by initial marking in future.

But there are new places available for initial marking which arrive after linking. For example if we link-up the net to place p_0, we need for this operation just to copy already the existed net (see figure 9).

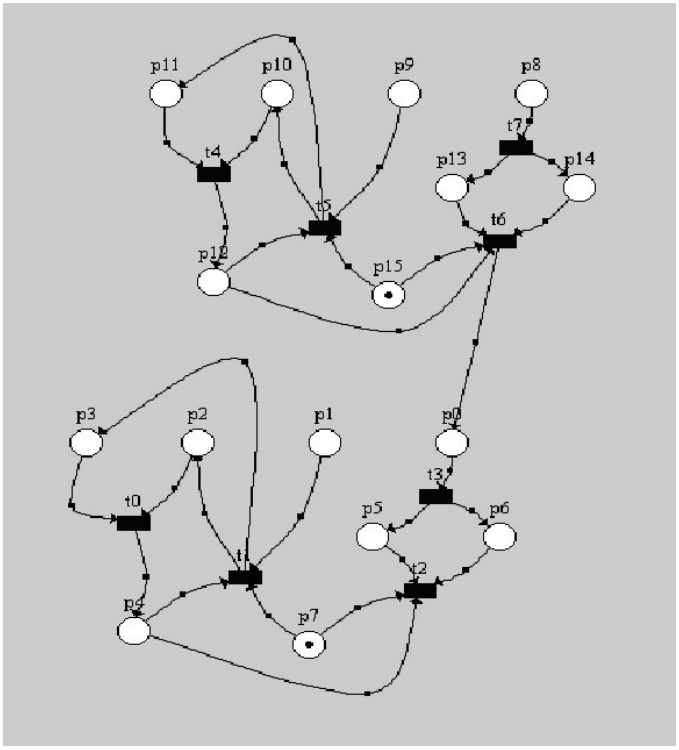

Fig. 9. Copying and linking the Petri net to increase the net complexity.

After that simple operation the transition t_2 remains decisive and new places ($p_8 ... p_{11}$) are available for an initial marking. Obviously transition t_6 plays the role of transition t_2 from lower net. Obviously if t_2 is executed, t_6 must be executed too and for this we mark the places as followed:

$$p_1 = 0, p_2 = 1, p_3 = 1, p_8 = 1,$$
$$p_9 = 0, p_{10} = 1, p_{11} = 1.$$

The key length grows up to 7 bits. Now for brute forcing we need 128 keys. Similarly linking-up nets for p_1, p_2, p_3 we will increase the key length up to 16 bits - there would be now possible 2^{16} keys.

But we can not similarly link-up the net to the p_1 place because for the reachability of t_2 we need it's empty

state. If we link-up the net for p_1 similarly to other places then it's empty state is reachable from the linked-up net by $15\ (16-1)$ different combinations. That is why we need to modify our linked up segment. As a variant we can use the following net shown in figure (10):

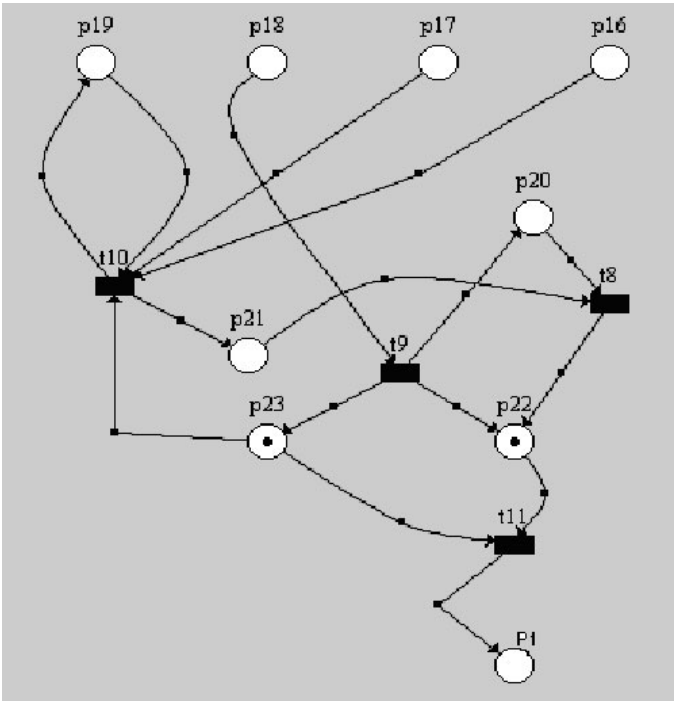

Fig. 10. Linked-up Petri net as a variant.

Here for unreachability of t_{11} by initial marking we put tokens to following places:

$p_{16} = 1, p_{17} = 1, p_{18} = 0, p_{19} = 1$

Whereas transition t_{11} must have lowest priority.

For 16-bit key we can build tables for the transitions. An important characteristic of such tables is the availability of changing places of rows or columns without violations in net structure - in that way we can complicate analysis for a software cracker.

IV. Difficulty of the Reachability Problem

The reachability problem - a reduced variant of the liveness problem - is defined by the following question:

> *Given a marked Petri net, m_0 being the initial marking and a marking m' - is m' reachable from m_0 ?* [4]

As far as problems of subsets and equalities for ensemble reachabilities of Petri nets are undecidable then maybe the reachability problem is undecidable too. To solve this problem several approaches are introduced by Mayr [6] and Sivaraman [9]. Other algorithms had been shown by Jancar [5] to be definitive wrong for solving such problems. One basic solution technique is to build a finite representation for the reachability set of a Petri net.

> *"As we can see, with reachability tree we can solve problems of safety, preservation and coverage. Regrettably, in common case we can't use it for solveing problems of B reachability and activity [...]". [8]*

Handorean [4] describes in a showcase how to build the reachability tree of a Petri net which also allows to build the entire state-space (see figure 11) and how to make the representation finite:

1) define ω number of tokens in a place then it is too big (plays the role of infinity)
2) when a new marking is equal to another marking on the path from the root node, we add it as a terminal node
3) a new marking x is grater than a marking y on the path from root, the components of y which are strictly greater than the corresponding components are replaced by ω (if $x > y$ then whatever is reachable from y is reachable from x too)

Which results into the following definitions:

- If the Petri net is k-bounded (max k tokens in a place), the reachable state-space is finite.
- If the Petri net is conservative and let k be the number of tokens in the net, the reachable statespace is finite (there is a finite number of ways we can partition k tokens among n places).
- If ω is anywhere in the reachability tree, the reachability set is not finite (and therefore cannot be bounded or conservative)

- If the reachability problem is solvable (possibly at a high cost) then the liveness problem is solvable.

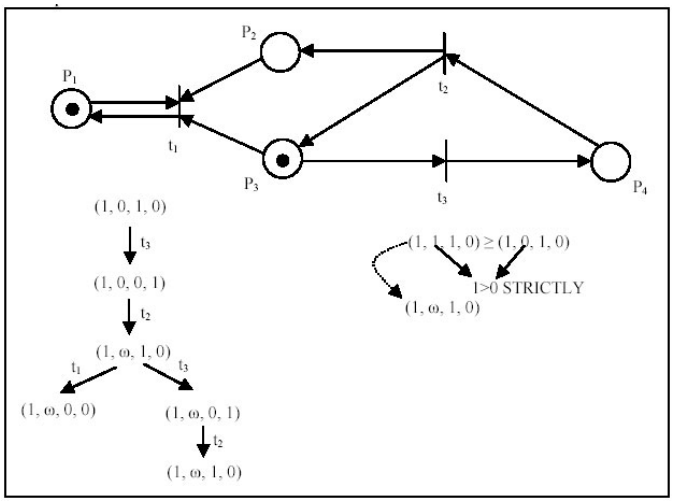

Fig. 11. Building the reachability tree of a Petri net (Image Source [4]

V. Discussion

Obviously standart methods of breaking this protection scheme do not work here. Unlike ”consecutive” protection schemes a software cracker never can guess which transition is decisive since upon wrong initial marking the decisive transition is not accessible. On the other hand, the software cracker may try to execute all transitions if he finds all places in any way. In our example the software cracker needs to try only 20 transitions instead of 2^{16} possible keys - he just has to put tokens in ALL places. It has not to be forgotten that our decisive transition has lowest priority - all tokens from input places disappear faster then it can check them. The software cracker could somehow find the transition with the lowest priority - for this we can enter thousand fictituous transitions with low priority levels and so on. But this is very inefficient for memory resources and performance.

We note that debugging of such program is complicated because the software cracker needs to debug all treads (transitions) simultaneously. This is nearly impossible. If we somehow paste protection code in different places of the body of protectable program we can protect from statical cracking too.

Even it is possible to solve the reachability problem, solving large Petri nets might be time consuming and inefficient to solve. This is one advantage for the suggested protection technique. One solution in preventing even high computation on such nets might be using unsolvable problems like given with M_1 and M_2 2 Petri nets whereas a possible question could be if $R(M_1)$ is a subset of $R(M_2)$ or $R(M_1)$ is equal $R(M_2)$. Problem with this is a possible unsolvable net which even prevents to work correct.

VI. Conclusions and Future Work

In conclusion the main goal of any protection scheme is hiding some information from other people. There is no protection which reaches it’s main goal. That is why we think that protection schemes development is useless and hopeless in a depth sense. However the introduced technique is a new method to increase the complexity of protection significant. Further investigations should focus on enhancing our algorithm and it’s implementation and to run several attacks on the protection scheme.

References

[1] DIJKSTRA, E.W.: *Co-operating sequential processes.* In Programming Languages, F. Genuys, Ed., 43–112, 1968.

[2] FISCHER, E.: *Petri Net Kernel*. Available at http://www.informatik.hu-berlin.de/top/pnk/index.html, 2004.

[3] HACK, M.: *DECIDABILITY QUESTIONS FOR PETRI NETS.* , 1976. Available at http://portal.acm.org/citation.cfm?id=889753&dl=ACM&coll=portal.

[4] HANDOREAN, R.: *Petrinets - Notes*. Available at http://userfs.cec.wustl.edu/ cs576/Notes/Petrinets.pdf, 2003.

[5] JANCAR, P.: *Bouzinae’s algorithm for the Petri net reachability problem is incorrect.* , 2000. Available at http://www.cs.vsb.cz/jancar/zb00.ps.

[6] MAYR, E.W.: *An algorithm for the general Petri net reachability problem.* Proceedings of the thirteenth annual ACM symposium on Theory of computing, 238–246, 1981. May 11-13, 1981, Milwaukee, Wisconsin, United States.

[7] PELEG, M., I. YEH R.B. ALTMAN: *Modeling biological processes using Workflow and Petri Net models.* Bioinformatics, 2001. Running title: Modeling Biological Processes using Workflows.

[8] PETERSON, J.L.: *Petri Net Theory and The Modeling of Systems.* PrenticeHall, 1981.

[9] SIVARAMAN, E.: *AN APPROACH FOR SOLVING THE GENERAL PETRI NET REACHABILITY PROBLEM DUALITY THEORY AND APPLICATIONS.* Available at http://www.okstate.edu/cocim/members/eswar/duality.pdf, 2004.

VII. Attachment

```
; ************************************************************************
                .586p
                .model          flat,stdcall
        option          casemap:none
                include         \masm32\include\windows.inc
        include          \masm32\include\user32.inc
                include         \masm32\include\kernel32.inc
                includelib      \masm32\lib\kernel32.lib
                includelib      \masm32\lib\user32.lib
; ************************************************************************
                .data
key             db              00h
P               db              0,0,0,0,0,0,0,1,0,0
box_title       db              "Congratulations!",0
box_mes         db              "You've crack this easy one...
                                 But what you say if there was 1000 threads?",0
box_title2      db              "Shit...",0
box_mes2        db              "Invalid key",0
; ************************************************************************
        .data?
ThreadId        dd              ?
; ************************************************************************
                .code
                CT              MACRO StartAddress
                push            ThreadId
                push            EBX
                push            EBX
                push            StartAddress
                push            EBX
                push            EBX
                call            CreateThread
                ENDM
_start:
                xor             EBX,EBX
                call            byte2bit
                CT              _T6
                ;invoke          SetThreadPriority, EAX, THREAD_PRIORITY_ABOVE_NORMAL
                CT              _T2
                CT              _T3
                CT              _T1
                CT              _T4
                CT              _T5
                ;invoke          SetThreadPriority, EAX, THREAD_PRIORITY_LOWEST
                CT              _T7
                jmp             $
_T3:
                mov             AL,P[4]
                add             AL,P[5]
                dec             AL
                dec             AL
                jnz             _T3
                mov             P[6],1
                mov             P[4],AL
                mov             P[5],AL
                jmp             _T3
_T1:
                mov             AL,P[3]
                dec             AL
                jnz             _T1
                mov             P[3],AL    ; 0
```

```
                mov             P[4],1
                jmp             _T1
_T4:
                mov             AL,P[0]
                dec             AL
                jnz             _T4
                mov             P[8],1
                mov             P[9],1
                mov             P[0],AL
                jmp             _T4
_T5:
                mov             ECX,0FFFFFFh
                loop            $ ; delay
                mov             AL,P[8]
                add             AL,P[9]
                add             AL,P[7]
                add             AL,P[6]
                sub             AL,4
                jnz             _T5

                invoke          MessageBox, 40h, addr box_mes, addr box_title, EBX
                jmp             _fin
_T2:
                mov             AL,P[2]
                dec             AL
                jnz             _T2
                mov             P[5],1
                mov             P[2],AL
                jmp             _T2

_T6:
                mov             AL,P[6]
                add             AL,P[1]
                add             AL,P[7]
                sub             AL,3
                jnz             _T6
                mov             P[3],1
                mov             P[2],1
                mov             P[1],1
                mov             P[7],AL
                jmp             _T6
_T7:
                mov             ECX,0FFFFFFFh
                loop            $
_patch:
                invoke          MessageBox, 40h, addr box_mes2, addr box_title2, EBX
_fin:
                invoke          ExitProcess, EBX
byte2bit:
                mov             AL, byte ptr [key]
push             EAX
                shr             AL, 3
                mov             P[3],AL
                pop             EAX
                push            EAX
                shr             AL, 2
                and             AL,1
                mov             P[2],AL
                pop             EAX
                push            EAX
                shr             AL, 1
                and             AL,1
```

```
mov             P[1],AL
pop             EAX
and             AL,1
mov             P[0],AL
ret
end             _start
```