Algorithm for Digital Watermark Generation in Executable Program Memory

Sergey V. Belim [1,2]
[1]Omsk State Technical University
11 Mira avenue, 644050, Omsk, Russia
[2]Siberian State Automobile
and Highway University
5 Mira avenue, 644080, Omsk, Russia
sbelim@mail.ru

Sergey N. Munko
Omsk State Technical University
11 Mira avenue, 644050, Omsk, Russia
munko_s@mail.ru

Abstract

The algorithm for embedding digital watermarks in the memory of the executable program is developed. The executable code RAM is a container for embedding data. A digital watermark exists in the program's memory for a limited time. The address of the embedded information in the RAM is random. The random address complicates the detection and detection of the digital watermark. The password is used to extract embedding information. The dynamic library is implemented for embedding, extracting and removing a digital watermark. The dynamic library is part of the authentication process. This prevents the removal of the digital watermark.

1 Introduction

Controlling the distribution of paid software is one of the current tasks. This problem is related to monitoring unauthorized copying of programs. Changes are made to the program during illegal distribution to destroy information about the license key. The task of developing algorithms for embedding hidden information into program code is current. This hidden program code allows you to track and identify illegal copies of programs.

There are two main types of digital watermarks in executable code.

1) Static digital watermarks.

2) Dynamic digital watermarks.

Static digital watermarks are implemented based on one of three approaches. [1, 2]

1) Additional functionality is embedded in the program and activated under certain parameters of program startup.

2) Additional data fits into the constants of the executable code (arithmetic or string).

3) Unused commands are added to the program ("dead code").

Dynamic digital watermarks are implemented as dynamic data structure in RAM [3]. The content of the embedded message is determined by the topology of the dynamic structure.

In: Sergei S. Goncharov, Yuri G. Evtushenko (eds.): Proceedings of the Workshop on Applied Mathematics and Fundamental Computer Science 2021 , Omsk, Russia, 24-04-2021, published at http://ceur-ws.org

Static digital watermarks are not resistant to attacks on application memory analysis and executable code distortion [4, 5, 6]. Complete destruction of static digital watermarks can be performed using UPX or ASPProtect. These disadvantages are absent from dynamic digital watermarks.

The task of embedding digital watermarks in program code faces several problems. These problems are not present in similar tasks for images or videos. The first problem is the selection the modifiable data. The container data is distorted when embedded. These distortions are small. The modified image differs little visually from the original image. This approach is not applicable when embedding hidden messages into an executable program. Any program is presented as data and executable commands for processing this data. Any change in the command code leads to incorrect operation of the executable code as a whole. The corrected program may not be compiled. Changing the data gives the same result. Any change to the constants that initialize the variables causes the program to run incorrectly. Algorithms to solve this problem are proposed in some articles.

One approach is to embed a message in NOP chains [7]. NOP chains are present in any executable code. NOP chains appear in the program when the code segments are aligned to the size of the memory pages. The authors suggest replacing NOP commands with some commands containing an embedded message. These commands should not make changes to the program and prevent compilation. The embedded message is retrieved by disassembling the executable code. Information about the position of the embedded message in the program code is used to extract it. This approach is not applicable when embedding messages into program code in a high-level language. The second approach uses program graphics to embed a message. This approach is not universal. It is not applicable to console applications and dynamic libraries.

Obfuscation is also used to embed messages in executable code [8].

The second problem of embedding digital watermarks into program code is that the program algorithm has a strict logical structure. The program structure facilitates the task of finding and eliminating the embedding message. Steganalysis of images does not correlate between data. Some geometric shapes are present in the image. Finding shapes in an arbitrary image is a difficult task to recognize images. A message embedded in the least significant bit of the blue component with low container occupancy and random pixel selection is not detectable. The program code has a clear logical structure. This logical structure can be recovered from the original or disassembled code. Analysis of the logic of the program code allows you to easily identify non-functional inserts and detect built-in messages. The data present in the program are constants for initializing variables. Constants have a small volume. Modification of constants is not allowed.

2 Digital Watermark Embedding Algorithm

We formulate the basic requirements for embedding a digital watermark into executable program code.

1. The digital watermark is not present in the source code of the program as a value of some constant. Static analysis of open or disassembled code easily detects such DWM.

2. Digital watermark is generated dynamically during program execution.

3. A digital watermark is generated at some stage of the program execution. The digital watermark generation step is secret information.

4. The digital watermark is destroyed at some point in the execution of the program. A digital watermark exists in the RAM for a period of time. At this time interval, the digital watermark can be extracted. This approach significantly complicates dynamic code analysis to detect a digital watermark.

5. The digital watermark depends on the key information. Key information is entered interactively by the user.

6. The digital watermark generation instructions are distributed throughout the source code. This requirement prevents the removal of the digital watermark from the source code.

7. A digital watermark is generated in RAM. The digital watermark must not occupy a continuous address space block.

8. The digital watermark is divided into blocks. These blocks are stored using some dynamic data structure. The block locations depend on the key information entered by the user.

The C++ programming language is used to embed a digital watermark (DWM) into RAM. Dynamic Link Library (*DWM.dll*) is implemented using it. This library describes all methods for embedding data into RAM.

The DWM embedding algorithm consists of several steps.

The algorithm operation is shown in Figure 1.

1. The user is authorized when the application starts.

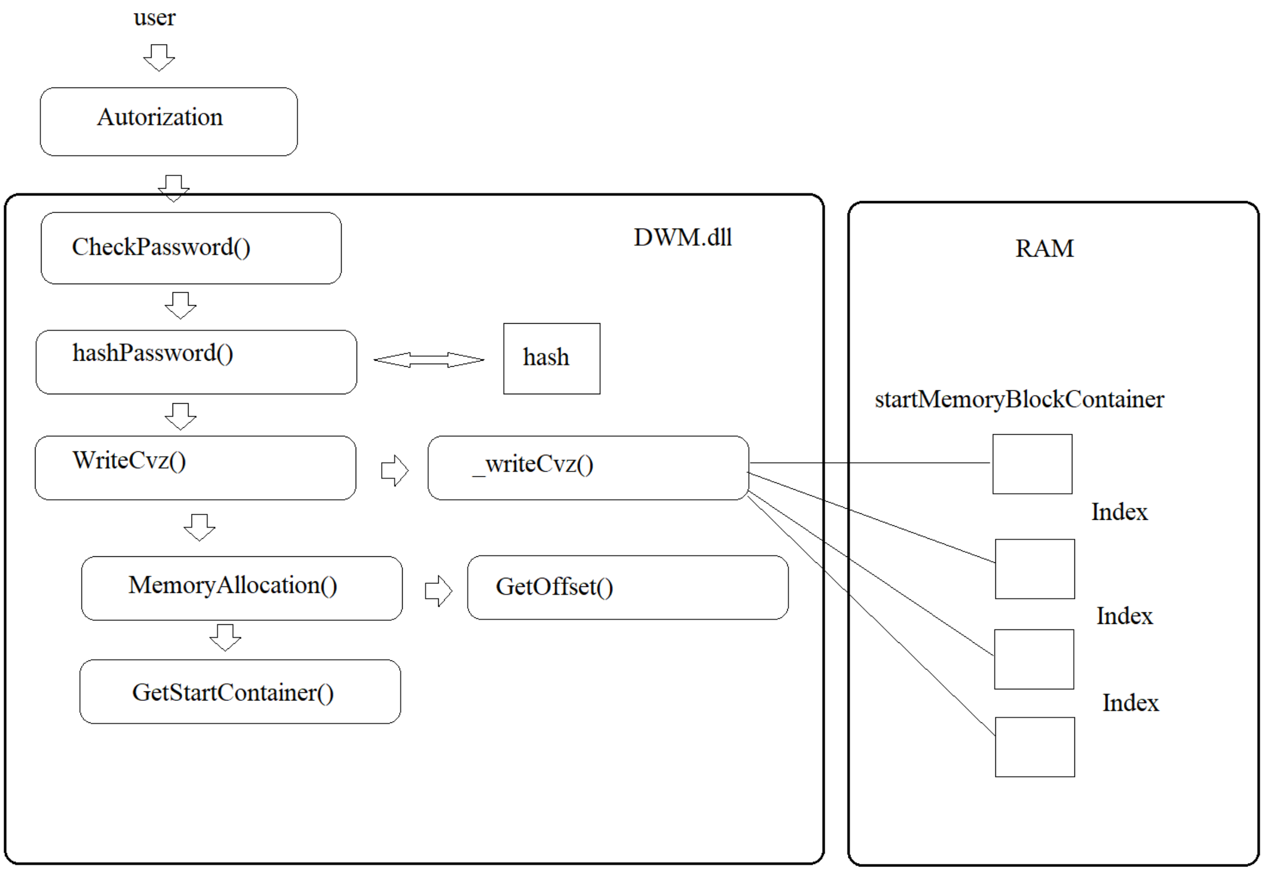

Figure 1: The algorithm operation.

2.The user enters a keyword instead of a password. The function $CheckPassword()$ from the $DWM.dll$ is called when a keyphrase is entered. This function calls the function $hashPassword()$, which forms the hash of the entered keyword.

3. The hash value for the keyphrase is compared to the hash value stored in the DWM.dll. If the two hash values are the same, the function $WriteCvz()$ is called. This function calls the function $MemoryAllocation()$, which forms a plurality of memory cells for storing DWM. All cells are 2 bytes in size. The number of cells is a system parameter. The total cell size exceeds the DWM size. This makes it difficult to find DWM in the program memory for an attacker. The address of the first allocated cell is stored in variable $startMemoryBlockContainer$. This variable has a global scope in the $DWM.dll$. The $startMemoryBlockContainer$ variable is used to read DWM from another program.

4. Function $WriteCvz()$ calls procedure $\_writeCvz()$. This procedure writes to RAM. The address of the first memory location for storing the first DWM symbol is determined by the function $GetStartContainer()$. This function calls the function $GetOffset()$, which finds an offset from the base address of the application stored in the variable $startMemoryBlockContainer$. The variable address used is subtracted from the base address of the application being developed.

5. The $Index$ is added to the address of the variable used to confuse the attacker. The $Index$ is calculated based on the start time of the application.

$$Index = (processCreation_time mod 100)/10.$$

$processCreation$ is the start time of the program process. We get a new offset of the base address for the variable.

6. We add the base address of the application to the new base address of the variable and get the address of the first cell for the DWM record.

7. DWM is treated as an array of characters. The first DWM character is written to the base address. The address of the next character is calculated by multiplying the previous address by the $Index$.

8. DWM is stored in RAM for a limited time. The DWM is removed from memory after a specified time interval. This complicates the search for DWM for the attacker.

This algorithm has some new properties.

1) The original DWM address is not stored explicitly inside the program (as a constant), but is calculated dynamically.

2) All subsequent addresses are calculated dynamically rather than stored in previous DWM cells.

3) A DWM is generated at a specific event.

4) The DWM is deleted after a period of time.

3 Digital Watermark Extraction Algorithm

A console application in the C++ programming language is implemented to read DWM. The application runs as part of a program that embeds DWM RAM. The program reads it, outputting DWM to the console.

The algorithm for reading DWM finds the process that wrote DWM to RAM.

webApi $FindWindowmethods()$ and $GetWindowThreadProcessId()$ are used for finding the identifier of process in which the record DWM was made. Method *OpenProcess*() is called. The process ID is passed to this method. The function *GetCvz*() gets the key. We define the start memory location address of the DWM write to obtain the key. The *webapi* $CreateToolhelp32Snapshot()$ method gets the base address of the application. We define the offset address *DWM.dll*, since the DWM embedding algorithm is implemented in it. Method $GetModuleBaseAddress()$ finds this address. The debugger in the DWM writer finds the base offset address of the *startMemoryBlockContainer* variable. The address will not change if you change the program structure. The offset address of the *startAdress* variable $startMemoryBlockContainer$ is calculated by the formula.

$$(dllAdress + 168672) + baseAddress.$$

dllAdress is the base address of $DWM.dll$. 168672 is an offset from the base application address of variable *startMemoryBlockContainer*. *baseAddress* is the base address of the application. The index calculation algorithm, relative to time, adds $Index$ to the variable *startAdress* and reads the DWM. The addresses of the following DWM characters are calculated by multiplying the base address by the resulting index.

4 Conclusion

The article proposes an algorithm for embedding dynamic digital watermarks into the RAM of the executable program. The proposed algorithm allows to securely hide the built-in message and ensure reliability of digital watermark extraction.

The software implementation the proposed method showed its high operability and stability, generated digital watermarks. The developed library has sufficient versatility and can be used in the implementation of secure software. The addresses sequence for the digital watermark in the memory is tested based on the developed library and a simple user authorization program. Random tests showed that the placement of digital watermark blocks does not have obvious patterns. Each block from the digital watermark requires independent detection when attacking the system. If the digital watermark is pre-encrypted, then it is not distinguishable from a random numbers sequence. Random placement and encryption ensure that the digital watermark is undetectable.

References

[1] I. Nechta. Robustness analysis for dynamic watermarks. *International Multi-Conference on Engineering, Computer and Information Sciences (IEEE, SIBIRCON)*:298–300, 2017.

[2] A.A. Zaidan, B.B. Zaidan, O.H. Alanazi et al. Novel approach for high (secure and rate) data hidden within triplex space for executable file. *Scientific Research and Essays*, 5(15):1965–1977, 2010.

[3] C. Collberg, C. Thomborson. Software watermarking: Models and dynamic embeddings. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*:311–324, 1999.

[4] M.D. Ernst. Static and dynamic analysis: Synergy and duality. *ICSE Workshop on Dynamic Analysis*:24–27, 2003.

[5] C. Linn. Obfuscation of executable code to improve resistance to static disassembly. *Proceedings of the 10th ACM conference on Computer and communications security*:290–299, 2003.

[6] A. Moser, C. Kruegel, E. Kirda. Limits of static analysis for malware detection. *Computer security applications conference*:421–430, 2007.

[7] R. Gabrys, L. Martinez, S. Fugate. How to swap instructions midstream: an embedding algorithm for program steganography. *HotSoS '20*, 25:1–2, 2020.

[8] K. Lu, S. Xiong, D. Gao. RopSteg: program steganography with return oriented programming. *Proceedings of the 4th ACM conference on Data and application security and privacy.*:265–272, 2014.