KYBERNETIKA — VOLUME *48* (2012), NUMBER 3, PAGES 494–501

C SOURCE CODE OBFUSCATOR

Lukáš Ďurfina and Dušan Kolář

Obfuscation is a process that changes the code, but without any change to semantics. This process can be done on two levels. On the binary code level, where the instructions or control flow are modified, or on the source code level, where we can change only a structure of code to make it harder to read or we can make adjustments to reduce chance of successful reverse engineering.

Keywords: obfuscation, source code, malware

Classification: 68N15

1. OBFUSCATION

The obfuscation is thoughtful process of modification with aim to hide information without causing any damage to this information. We know several types of obfuscation and they are shown in the following text.

1.1. Types of Obfuscation

The general division dictates two main areas:

- binary files obfuscation
- source files obfuscation.

Another division can be make according to a purpose:

- hide new algorithm, technology (protect intellectual property)
- hide well-known algorithm, technology (prevent unwanted detection).

In some point of view, we can say that obfuscation wants to achieve security through obscurity, what can be enough only in some particular cases.

2. OBFUSCATION OF BINARY FILES

The base for this obfuscation is knowledge of an instruction set of aimed architecture, because it is natural that obfuscator for ARM executable would damage executable for the x86 architecture or other incompatible platform. The advantage is that well-written obfuscator can be usable for more executable formats, if exist, for the given architecture. It is common to see obfuscator for both PE and ELF format on the x86 architecture [7].

We can take a look on the most used techniques [4]:

- Dead-code insertion

  Idea is same as inserting instruction `NOP`, but single instruction would not be very helpful, because it can be simply filtered. Dead code does nothing useful at all, it only decreases performance and confuses the code. It can consist of various complex algorithms, which unbend attention from real object of the inspection.

  Example:
  ```
  push eax
  push ebx
  push ecx
  ;some magic with eax, ebx, ecx
  pop ecx
  pop ebx
  pop eax
  ```

- Code transposition

  It is based on a finding of independent pieces of codes, and their mutual exchange. Transposition can be made on two instructions, but it can also be done on whole blocks. It depends on the skills of author how precisely he can determine independent blocks of code. Another way of code transposition is adjusting jumps and calls, and reassembling blocks of such code.

  Example:
  ```
  mov eax, [ecx]                      mov ebx, 10
  mov ebx, 10                         mov eax, [ecx]
  mul ebx                             mul ebx
  ```

- Register realignment

  It is simple method, when we exchange the certain number of registers. The code works with other registers, so the bytes in binary code, which represent used registers, are different, but an algorithm is still the same one. In another words, we create different admissible permutation of registers for the given code.

  Example:
  ```
  mov eax, ebx                        mov ecx, edx
  xor ecx, ecx                        xor eax, eax
  test eax, ecx                       test ecx, eax
  ```

  The important note is that there are some restrictions. It definitely would not be a good idea to exchange register `esp` on the x86 architecture, because it would cause a corruption of stack, what uses to end up with the crash of application.

- Instruction substitution

  Instruction set of the x86 architecture is very wide, so the single action can be performed by more combinations of instructions [3]. This fact is used for substitution technique. We can distinguish 3 subcategories according to the change of code size:

  - code expansion - new code is formed by more instructions than original

```
add eax, 4                        add eax, 2
                                  add eax, 2
```

  - code shrinking - new code has less instructions than original

```
add eax, 100                      mov eax, 1
mul 0
inc eax
```

  - code alternating - new code has same size as orginal, following three blocks do the same

```
mov ebx, 0                        xor ebx, ebx
mov eax, eax                      mul 1

sub ebx, ebx
add eax, 0
```

A very popular approach is exchanging instructions, which have different semantics, but with clever updates they have the same result. We present it on the replacing of `push` and `call`.

Examples:

```
push eax                          sub esp, 4

                                  mov [esp], eax

call sub_count                    push 401020

                                  jump sub_count
```

In the first example, we decrease stack pointer, and after that we store value from `eax` to stack. If we use in the following code `pop` the beginner could be confused due to no `push`. The second example uses the fact how the instruction `call` works, it stores the address, where the control should be returned after finish of function. This approach has a great advantage. You can set an arbitrary address for continuing after return from function.

The binary form of files provides another opportunities for code obfuscation, the good example is function call dispatching, what is nicely implemented in PEScrambler [2]. The technique is based on the redirections of all internal and external function calls to a single function, which acts as the dispatcher. The result is that all instructions `CALL` has the same operand and the reverse engineer does not know which specific function is called from the dispatcher.

3. OBFUSCATION ON SOURCE CODE LEVEL

Source code obfuscation can be divided into two types according to the result of the obfuscation:

- obscure source code to make it harder to read and understand
- obscure compiled executable to make harder to understand its disassembly.

In this paper we are interested in the second type.

Editing source code is essentially different from editing binary code. There is no possibility to use some introduced techniques from the previous section, for example register realignment.

On the other side, we can easily hide the data in program. We can imagine that all strings can be written in an encrypted form, and in the moment of use there will be called a decryption function, so the base function will get the data in correct form, but in the source and also in the data section of executable, the strings will be illegible. The encrypted form and the decryption function can be changed for each compilation, so every released version could have the different data section and also different code for the decryption routine. This method could be utilized by botnet owners for better hiding of bots. The majority of bots support self update and by this way they could be updated regularly by new version.

We can also approximate function call dispatching, the function calls will not be redirect to single dispatcher, but from the code it would not be easy to recognize which function is called. The disadvantage is that it cannot be applied for all functions, but only for the linked functions from the extern libraries. On the other hand, this method can hide the imports of variables from such libraries too. The trick is done by using WINAPI functions `LoadLibrary` and `GetProcAddress` [6]. The following example obfuscates the call of `CreateFileA`.

```
HINSTANCE hDLL = LoadLibrary(‘‘Kernel32.dll"); if (hDLL) {
  fCreateFile = GetProcAddress(hDLL, ‘‘CreateFileA");
}
```

After successful running of that code, we can use `fCreateFile` the same way as WINAPI function `CreateFileA`. Still it is not so good, in spite of fact that we do not see direct call to WINAPI function, the function name is stored in the data section as a string. Anyway this is solved by the string encryption, which encodes both strings in that example, so the name of function and library will be completely hidden. The problem is recognition of those functions and loading the correct library for each one. The solution is a database of libraries and corresponding functions, which would be complex and has to be religiously updated.

Technique	Binary code obfuscation	Source code obfuscation
Dead-code insertion	✓	✓
Code transposition	✓	✓
Register realignment	✓	×
Instruction substitution	✓	×
Function call dispatching	✓	∼
Strings encryption	×	✓

Tab. 1. The comparison of binary and source code obfuscation.

3.1. Comparison with binary obfuscation

Table 1 presents the summary and compares the both types of obfuscation. The sign ✓means that the method is possible by the given type, the sign × means impossibility and ∼ is sign for partially possible method.

3.2. Obfuscating compiler

For the future research, we suggest more powerful solution, the obfuscating compiler. The strength is given by possibility of applying all mentioned techniques with the advantage that compiler gives the higher probability of producing correct executable.

The idea goes from the fact that if you compile the the source code by the same compiler with unchanged flags, you will get each time the same output executable. This is easily verified by comparing the disassemblies. We can imagine the module for GCC, which will work as add-on for code generator that will ensure providing different executable each time. The generation of code will be nondeterministic process after this modification, so still it would be possible to generate two same executables, but the probability would be very small. The measure of obfuscation, which is important for the performance of compiled program, would be configured in the same way as an optimization. Such a compiler would be more powerful than metamorphic engines, but the positive information is that it would be difficult to integrate it into malware due to its size and the presence of source code.

4. EXPERIMENTAL APPLICATION

According to pointed approaches the simple obfuscator for C source code was implemented. This tool supports two methods:

- string encryption
- dead-code insertion.

String encryption is based on Caesar cipher [8], so it is not very complicated, but for our purposes it is enough. Every string, which can be encrypted by this implementation, is exchanged by a call of decryption function with encrypted string as a parameter.

Example:

```
printf(``Hello"); printf(csoob_decrypt(``Lipps"));
```

Dead-code insertion is based on an inserting mathematical operations – addition, subtraction, multiplication, and division. The most of nowadays compilers can detect dead code and remove it from a final generated executable. This removal can be prevented by using the keyword `volatile` with variable declaration [1]. This keyword is used for appointing variable for hardware access or inter-thread communication, what prohibits the optimization on such variable.

Example:

```
volatile int __csoob_530 = 2820;
__csoob_530 = __csoob_530 - 5347;
__csoob_530 = __csoob_530 * 28033;
__csoob_530 = __csoob_530 / 25;
```

The both of used methods could be improved. String encryption is now allowed only for the first parameter in single function call. This is caused by the fact that we cannot use dynamic allocation of memory, because the deallocation of these strings would be very complex problem. The restriction for only the first parameter is given by using a global variable for returning the result. So if we used decryption function on more than single parameter, the parameters values would be corrupted. The next version will be able to encrypt more parameters. It will be implemented by more decryption functions, so each parameter will have own function with own global variable.

Insertion of dead-code will be improved by including more difficult codes. We could insert computationally complex code that is never called, so the performance is not decreased, but the disassembly is definitely longer and more confusing.

Example:

```
volatile int __x_0 = 0, __x_1 = 0;
if (__x_0 || __x_0 + 1 == __x_1)  __x_1 = qsort();
```

4.1. Results

The created obfuscator was tested with various malware source codes, and the results are shown in Table 2. Obfuscated malware was partially tested for correctness. Each executable was run to verify if the changes of source code did not cause a segmentation fault or another runtime error. This test passed without any error. We did not test if the behavior was not changed, because it could be different for each run also without any recompilation.

The first column contains name of malware, the following columns resume the number of detections in that order: unmodified original code, obfuscated (A) (string encryption and dead-code insertion), and stronger obfuscated (B) (string encryption and higher measure of dead code). Inserting of dead code implies larger executables. In tested examples, for the test case (A) the size was increased by approximately 15 %, and stronger obfuscation (B) increases size by approximately 30 %. Also a time of compilation was longer, it takes about 50 % more for (A) and about 100 % more for (B), but malware is usually small in the source code size, so this higher time consumption is not critical,

Malware name	Original code	Obfuscated (A)	Stronger obfuscated (B)
Batzback	22/43 - 51.2%	13/43 - 30.2%	10/43 - 23.3%
Branko	10/43 - 23.3%	1/43 - 2.3%	0/43 - 0%
Cairuh	22/43 - 51.2%	18/43 - 43.9%	14/43 - 32.6%
Darkness IRC bot	14/43 - 32.6%	5/42 - 11.9%	4/43 - 9.3%
Hexbot	16/42 - 38.1%	14/43 - 32.6%	7/43 - 16.3%
JrBot	27/43 - 62.8%	18/43 - 41.9%	14/43 - 32.6%
Kbot	20/43 - 46.5%	12/43 - 27.9%	11/43 - 25.6%
lolworm	16/43 - 37.2%	11/43 - 25.6%	11/43 - 25.6%
Littlepain	14/43 - 32.6%	6/43 - 14%	4/43 - 9.3%
Akbot	25/43 - 58.1%	15/43 - 34.9%	14/43 - 32.6%
Mydoom	20/43 - 46.5%	11/43 - 25.6%	10/43 - 23.3%
Newstar	24/43 - 55.8%	20/43 - 46.5%	18/43 - 41.9%
MSBlaster	20/43 - 46.5%	17/43 - 39.5%	17/43 - 39.5%
Hunatchab	10/43 - 23.3%	0/43 - 0%	0/43 - 0%
Netsky	15/42 - 35.7%	13/43 - 30.2%	12/43 - 27.9%
Matrix	17/43 - 39.5%	5/43 - 11.6%	1/43 - 2.3%

Tab. 2. The results of experiment with malware.

because it was in all cases shorter than 5 seconds. For testing, we used the web service VirusTotal[1]. This service provides scanning of the uploaded file by 43 anti-virus applications.

The good result was achieved with *Branko* and *Hunatchab*, which were not detected after stronger obfuscation at all. Also malware *Win32.Matrix* was obfuscated very successfully, the count of detections was reduced from 17 to only 1. In the sum, the executables from original code were marked as malware 292 times, after obfuscation (A) there were 179 detections and finally, malware with stronger obfuscation (B) was detected in only 147 cases, so the rate of detection was lowered by approximately 50 % of the original amount.

5. CONCLUSION

Each type of obfuscation can have good results for hiding the real object of code. The positive information is that they are independent, so it is possible to use both on a single program. Firstly, the source code is obfuscated, and after compilation there is used binary obfuscation on executable.

As it was demonstrated by the experiment, this could be handily used by malware creators and the antivirus software vendors are not prepared for such an option. There is a need for creation of a generic decompiler, which would be able to recognize similar behaviour of executables regardless of used arbitrary type of obfuscation.

Also, antivirus software vendors should enhance a recognition of dead code, because, at this time, such a code can break pattern matching, which is used for malware detection

[1]Free Online Virus, Malware and URL Scanner `http://www.virustotal.com/`

and they should put a lower significance to data in data sections, because such data can be modified in various ways, and their correct representation can be gained in the moment of use.

ACKNOWLEDGEMENT

This work has been supported by the research funding TAČR TA01010667, and by the Research Plan No. MSM 0021630528.

(Received June 4, 2011)

REFERENCES

[1] Free Software Foundation, Inc.: Volatiles – Using the GNU Compiler Collection. 2010, `http://gcc.gnu.org/onlinedocs/gcc/Volatiles.html`.

[2] N. Harbour: Advanced Software Armoring and Polymorphic Kung-Fu. DEFCON 16, 2008.

[3] Intel: Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference. 1999, `http://download.intel.com/design/intarch/manuals/24319101.pdf`.

[4] A. Karnik, S. Goswami, and R. Guha: Detecting Obfuscated Viruses Using Cosine Similarity Analysis. Modelling Simulation, 2007.

[5] D. Low: Protecting Java code via code obfuscation. In: Crossroads – Special Issue on Robotics, 1998.

[6] Microsoft: MSDN Library. `http://msdn.microsoft.com/en-us/library/ms123401.aspx`.

[7] A. Moser, Ch. Kruegel, and E. Kirda: Limits of static analysis for malware detection. In: Computer Security Applications Conference, 2007.

[8] Ch. Savarese and B. Hart.: The Caesar Cipher. 1999, `http://www.cs.trincoll.edu/~crypto/historical/caesar.html`.

Lukáš Ďurfina, Faculty of Information Technology Brno University of Technology, Božetěchova 1/2, 612 66 Brno. Czech Republic.
e-mail: idurfina@fit.vutbr.cz

Dušan Kolář, Faculty of Information Technology Brno University of Technology, Božetěchova 1/2, 612 66 Brno. Czech Republic.
e-mail: kolar@fit.vutbr.cz