International Journal of Engineering Science Invention
ISSN (Online): 2319 – 6734, ISSN (Print): 2319 – 6726
www.ijesi.org Volume 3 Issue 5‖ May 2014 ‖ PP.01-10

Source Code Obfuscation: A Technique for Checkmating Software Reverse Engineering

[1]Edward E. Ogheneovo and [2]Chidi K. Oputeh,
[1,2,]Department of Computer Science, University of Port Harcourt, Port Harcourt, Nigeria

ABSTRACT : *With the widespread use of personal computers and advent of the Internet technology, computer security has suffered a lot of setbacks. Computer users especially hackers these days are finding ways to get access to certain information on the Internet which ordinarily they suppose to have use passwords to get access to. The result is that these scrupulous users bypass username and passwords to access information that are not well protected. One such problem is software cracking. Attackers and hackers bypass protection copyright, download software and use them without permission. Software cracking is the process of bypassing the registration and payments options on a software product to remove copyright protection safeguards or to turn a demo version of software into a fully function version without paying for it. In this paper, we used source code obfuscation technique to checkmate software reverse engineering which is one of the techniques hackers often used to crack software. In this technique, the original code is mangled such that the resultant code, the obfuscated code, is difficult to analyze by the hackers, although; the code retains its originality. Our result shows that the obfuscated code cannot be easily decoded. Hence, it prevents hackers from cracking the code.*

KEYWORDS: Software obfuscation, reverse engineering, software cracking, obfuscated code

I. INTRODUCTION

Security is a very crucial aspect of computer science and it plays a major role in the advancement of Information Technology especially with the growing use of computer software for e – commerce, e – banking and for governance in all areas of life where national security is a very critical issue [10]. It is therefore relevant to study this area with increased efforts so that software produced and made - available for the general public can be more secured and reduce the possibilities of being attacked by crackers or hackers [7] [8]. This will give more confidence to the software developers and manufacturers to continue to create and develop software knowing that their software are saved and cannot be hacked or cracked. However, if hackers decide to hack the software, they can be sure that their software can stand the test of time.Code obfuscation [3] [11] focuses on the protection of a software program and the content that the program protects. There have been billions of dollars spent each year by the industries especially for software piracy and digital media piracy. The achievement of the content/software security in a huge segment is based on the ability of protecting software code against tampering and identifying the attackers who issue the pirate copies [6] [9]. Linn and Debray [12] concentrates on the attacker identification and forensic examination. The author discussed about a proactive detection approach for defeating an on-going attack before the cooperation has occurred. The author also describes another detection approach for post compromise attacker identification. Especially, the author takes into account the real world scenarios where the application programs connect with their vendors so often, and a discovery of attack can bar a hacker from further business.Code obfuscation focuses to protect code against both static and dynamic study and there exists another approach to protect against code analysis, namely self-modifying code [4]. This approach provides the opportunity to create code at runtime, rather than changing it statically. Practically, self-modifying code is highly restricted to the monarchy of viruses and malware. Yet, some publications regard self-modifying code as an approach to protect against static and dynamic analysis. Madou et al. [13] [14] for instance regard dynamic code generation. The author proposed an approach where functions are generated earlier to their first call at runtime. Moreover, clustering is presented in such a way that a general template can be utilized to generate each function in a cluster, carrying out a least amount of alterations. The decryption at runtime technique is equal with code generation, apart from the fact that the decryption key can depend on other code. Moreover, it lessens re-encrypting the viability of code during execution [15] [16]. However, the technique does not clearly protect a function template after the function executed.

II. REVERSE ENGINEERING PROCESS

The reverse engineering process [2] begins by extracting detailed design information, and from that extracting a high-level design abstraction. Detailed (low-level) design information is extracted from the source code and existing design documents. This information includes structure charts and data descriptions to describe processing details. The high-level design representation is extracted from the recovered detailed design and

expressed using data-flow and control-flow diagrams. The recovered design is the same as the extracted design. The procedure steps are discussed below. Figure 1 summarizes the procedure.

- **Collect information:** Collect all possible information about the program. Sources of information include source code, design documents and documentation for system calls and external routines. Personnel experienced with the software should also be identified.
- **Examine information:** Review the collected information. This step allows the person(s) doing the recovery to become familiar with the system and its components. A plan for dissecting the program and recording the recovered information can be formulated during this stage.
- **Extract the structure:** Identify the structure of the program and use this to create a set of structure charts. Each node in the structure chart corresponds to a routine called in the program. Thus the chart records the calling hierarchy of the program. For each edge in the chart, the data passed to a node and returned by that node must be recorded.
- **Record functionality:** For each node in the structure chart, record the processing done in the program routine corresponding to that node. A Program Design Language (PDL) can be used to express the functionality of program routines. For system and library routines the functionality can be described in English or in a more formal notation.
- **Record data-flow:** The recovered program structure and PDL can be analyzed to identify data transformations in the software. These transformation steps show the data processing done in the program. This information is used to develop a set of hierarchical data flow diagrams that model the software.
- **Record control-flow:** Identify the high-level control structure of the program and record it using control-flow diagrams. This refers to high-level control that affects the overall operation of the software, not to low-level processing control.
- **Review recovered design:** Review the recovered design for consistency with available information and correctness. Identify any missing items of information and attempt to locate them. Review the design to verify that it correctly represents the program.
- **Generate documentation:** The final step is to generate design documentation. Information explaining the purpose of the program, program-overview, history, etc, will need to be recorded. This information will most probably not be contained in the source code and must be recovered from other sources [2].

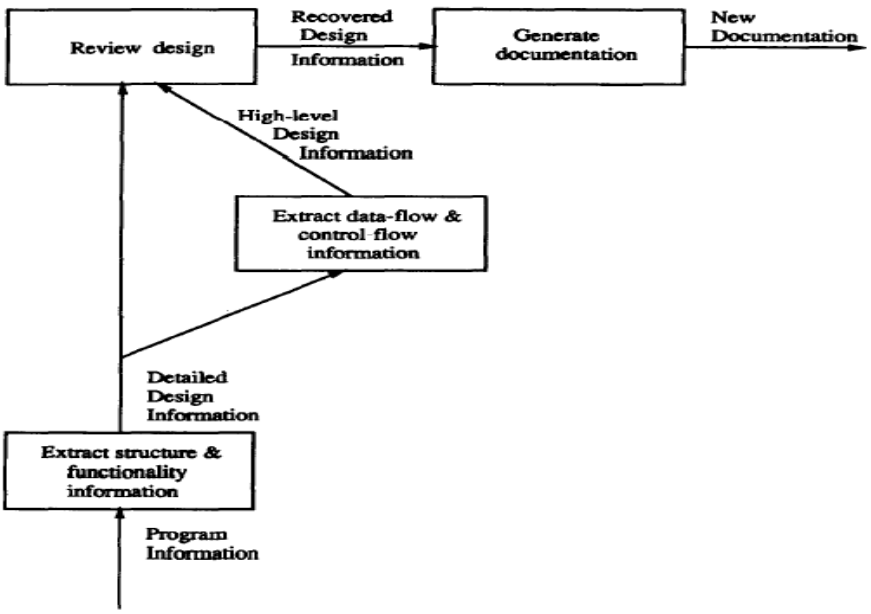

Fig. 1: Reverse engineering procedure
Source: Chikofsky and Cross [2]

Software reverse-engineering, or simply software analysis, mainly consists of investigating software. Depending on the intent of the attacker, one can extract hidden algorithms, secret keys, and other information embedded in the software. While analysis can be the actual goal of the attacker, it often is a precursor to tampering. Hence, hindering analysis also hinders tampering. Being able to have access to all the features of cracked software

implies that an attacker has full control over the host system and can therefore perform static and/or dynamic analysis techniques at will.

Static Analysis: This technique is applied on non-executing code and comprises static disassembling and subsequent static examination steps. Disassembling code is usually done using either linear sweep or recursive traversal. Linear sweep simply scans over the code, disassembling instructions, assuming that every instruction is followed by another instruction. GNU's(UNIX – like Operating system) gdb(GNU debugger) [16] uses this technique. Recursive traversal takes control flow into account. However, as some branches are input-dependent, usually not all target addresses can be statically derived and disassembled. Linear sweep is easily fooled by inserted data bytes [12], while recursive traversal often gives incomplete results. As a reaction, Schwarz et al. [16] propose to use a combined approach. Additionally, Krügel et al. [11] treat every memory address as a potential start of an instruction. Consequently, they filter out overlapping assembly results based on a control flow graph based approach. Additionally to disassembly, a decompilation step could map low-level code to more high-level constructions [3]. These high-level structures might facilitate human inspection more than abstract assembly listings. For certain languages such as Java or .NET, it is easy to decompile bytecode into source code [15]. Even if static analysis cannot derive all control flow, it is considered more complete than dynamic analysis as it examines all possible paths while dynamic analysis only considers the executed path. Figure 2 gives a schematic overview of the disassembly and decompilation phase for reverse engineering code. Recovered high-level code aims to approximate the original source code.

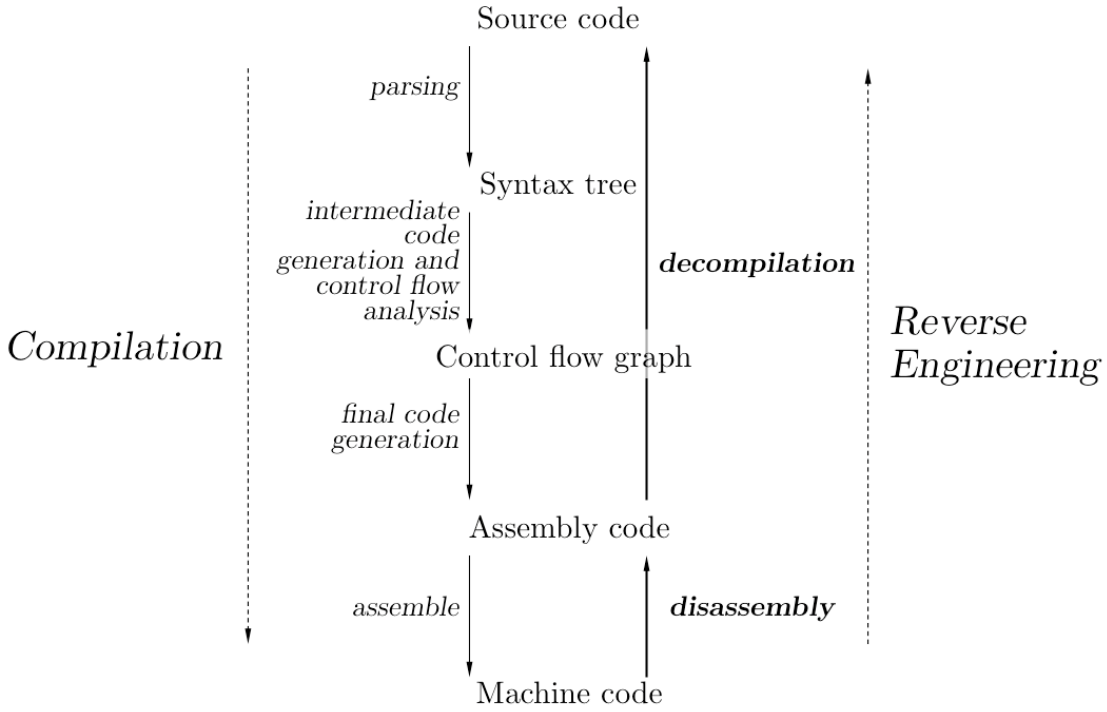

Fig. 2: Different reverse engineering stages, represented as the inverse of a compilation process.
Source: http://www.cosic.esat.kuleuven.be/publications/thesis-199.pdf

Dynamic analysis: Dynamic techniques are performed on executing code. This involves tracing of executed instructions, register contents, data values, etc. While this type of attack is more powerful than a static attack, it may be more time consuming or more complex. First, it requires a platform similar to that of the target code. Secondly, a program might be equipped with anti - debugging techniques hindering dynamic analysis. Debugging and emulation belong to the dynamic analysis. Commercial debuggers include SoftICE [1] and IDA Pro [5].

III. METHODOLOGY

Code obfuscation attempts to transform a program into an equivalent one that is more difficult to manipulate and reverse engineer. Code obfuscation attempts to make the task of reverse engineering a program daunting and time consuming. This is done by transforming the original program into an equivalent program, which is much harder to understand, using static analysis.
Considering our obfuscation mechanism, we define obfuscation as follows:

Let *T(P)* be program, transformation of program *P*. *T* is an obfuscating transformation, if *T(P)* has the same observable behavior as *P*. In addition *T* must follow conditions:

- if program *P* fails to terminate or terminates with an error condition, then *T(P)* may or may not terminate
- otherwise *P* terminates and *T(P)* must terminate and produce the same output as *P*

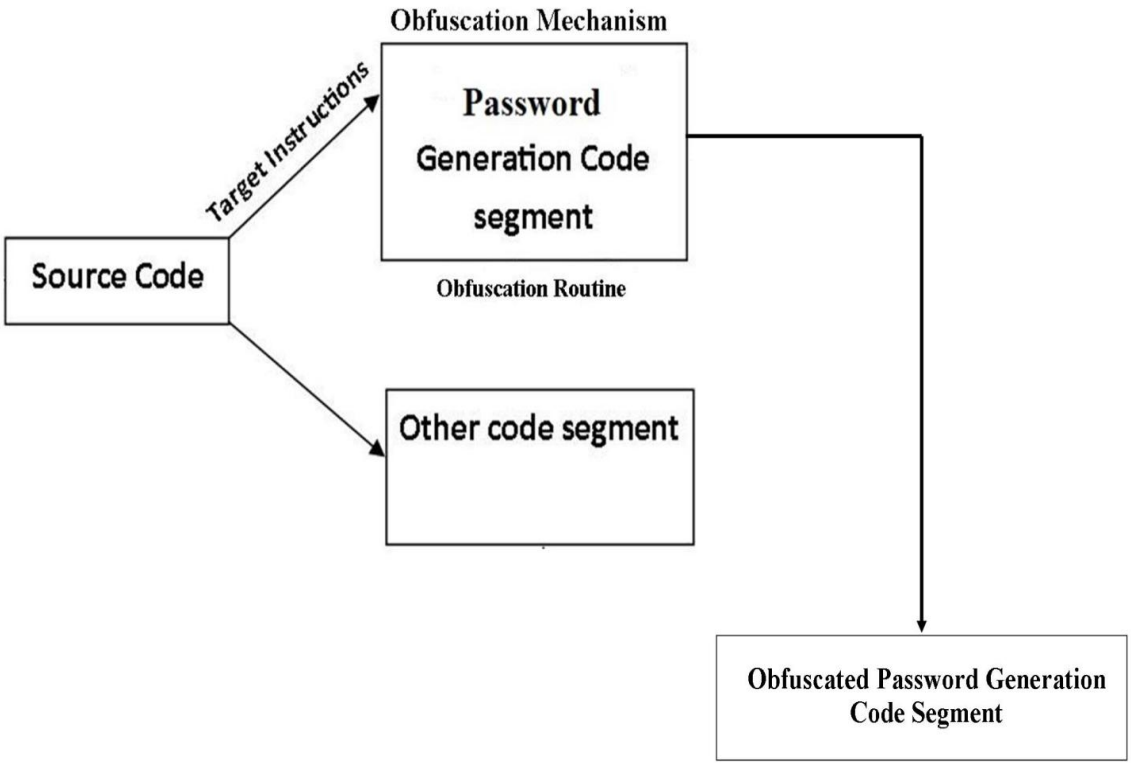

Fig. 3: Obfuscation Mechanism Module

In this mechanism, the encrypted serial number generation code segment and the other code segments are, both, subjected to obfuscation. This is done by writing an obfuscation routine within the source code that will be responsible for applying the transformation on the source code itself. Since the entire source code will be obfuscated, it is very difficult for any cracker or hacker to crack or have access to the internals of the program. Any cracker will lose the interest to crack any program that our obfuscation mechanism is applied on. This is because the sight of the program code after been obfuscated is mangled that it does not follow the normal chronological order of writing a program neither does it look like any other program seen around.

The technique involves lexical, control and data transformations. Lexical transformations alter the actual source code, such as C code. This transforms the original source code into a lexically equivalent form by mangling names and scrambling identifiers. Such transformations make it a daunting task to reverse engineer a program. Control transformations alter the control flow of the program by changing branch targets to an ambiguous state. The code for the program is shuffled such that the original branch targets are no longer correct. During this shuffling, the new targets are calculated, and code is inserted in place of the old branch instruction to acquire its new target address. Data transformations rearrange data structures such that they are not contiguous. Data can be transformed all the way down to the bit level.

Our obfuscation mechanism utilizes an algorithm for the obfuscation of the serial number generation code segment and the other code segment. The pseudo code is written below:

Pseudo code 1: Obfuscation Mechanism

```
[1]  Set string literals to array of numbers
[2]  Extract integer address of the array
[3]  Add index variable to the arguments list of a recursive function
[4]  Extract 3 lower case l's out of the array and insert a Boolean statement
[5]  IF the index variable is at the proper position of the array THEN
[6]  print 'l'
[7]  END IF
```

[8] Add a second index to keep track of l's that is printed
[9] Subtract the number from the index variable to access the real array element
[10] Shift around the location of the conditional operators
[11] **WHILE** switching some of the integers for character literals
[12] **RENAME** a few of the variables
[13] **END WHILE**
[14] **RENAME** functions to something similar to variable names
[15] **GET RID** of all unnecessary argument type specifiers
[16] **MANIPULATE** the formatting to obscure the flow of conditional operators and mask distribution between function name and main

Our obfuscation mechanism utilizes an algorithm for the obfuscation of the serial number generation code segment and the other code segment. This is shown in algorithm 1.

Algorithm 1: Obfuscation Mechanism

[1] Init sl = strings[ ]
[2] Init ia = address of sl
[3] Init al = argument list of recursive function
[4] **FOR** x = 1 to length of al
[5] al = al + sl[x]
[6] **END FOR**
[7] Insert 3 lower case l's into a Boolean statement
[8] **FOR** x = 1 to length of al
[9] **IF** x is at the proper position of sl, then
[10] print 'l'
[11] **END IF**
[12] Init al[x+1] = printed "l"
[13] **SUBTRACT** n from x to get element of sl
[14] **SWAP** conditional operators
[15] **WHILE** (Swapping between integers and characters)
[16] **RENAME** variables
[17] **END WHILE**
[18] **END FOR**
[19] **RENAME** functions to look like variable names
[20] **ELIMINATE** argument type specifiers

IV. RESULTS AND DISCUSSIONS

We cracked software (crackme.exe) that has a serial key authentication attribute. We want to attempt to use the same cracking rules to crack the hybridized self - modifying mechanism. In doing this, we will have some program generated outputs to show the results as we undergo the cracking processes. The hybridized self - modifying code is compiled and the executable named ObfusSec.exe is generated at compilation time. We start by running the source code to check the serial number parameters. Figure 4 shows the assembly language representation of the source code used for implementing this work. The figure shows the offset location of the jmp call. When the crackme.exe is run, it usually prompt the user to enter a serial number.

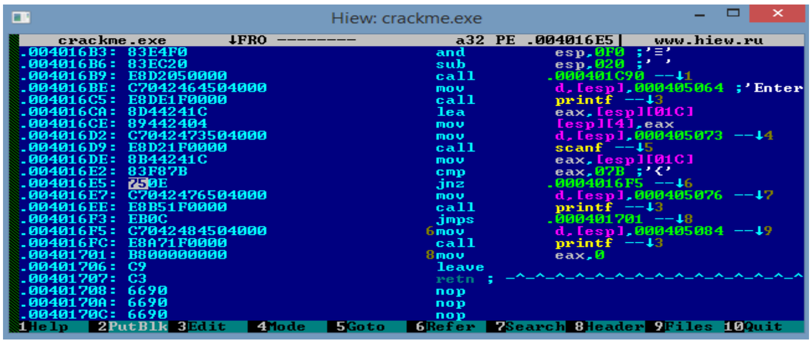

Fig. 4: Hiew displaying the Offset location of the jmp call

In Figure 5, the C++ source code of our model is opened using the Code::Blocks 13.12 IDE. We run the code and observed the behavior it displays. The resulting code is not readable and shows that the code has been obfuscated and converted to the form that is difficult to understand by someone intending to crack the code.

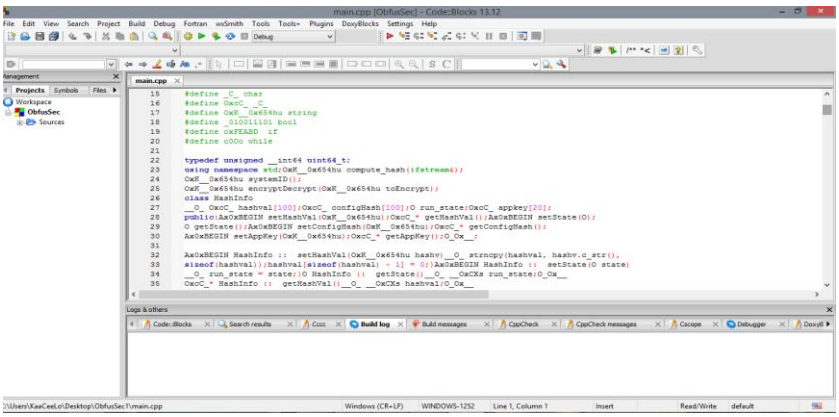

Fig. 5: The source code of ObfusSec.exe

In Figure 6, the C++ source code of our model is built and compiled. Then, a screen display showing an authentication scheme pops up. You are required to enter your key. If a wrong key is entered, it will display "invalid" showing that the key is not the correct one. This way the software cannot be cracked. However, if a key different from the correct key is entered and the word "invalid" is not displayed, it means that the code has been cracked.

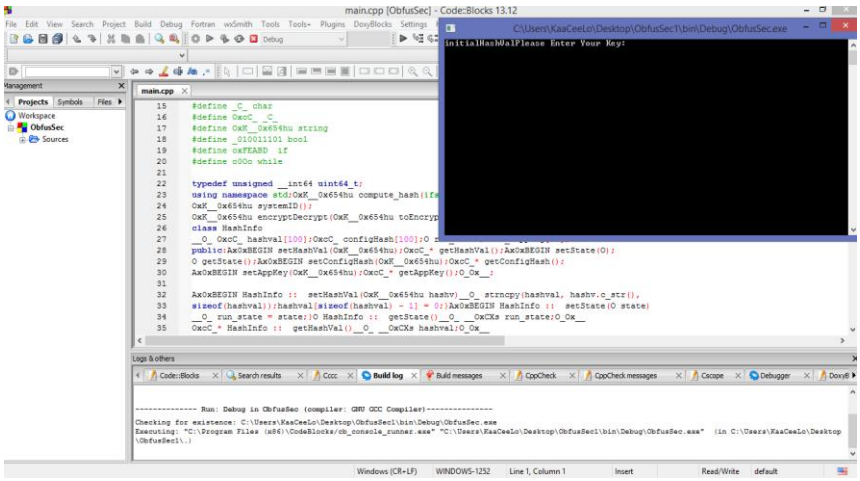

Fig. 6: Screen display to enter key

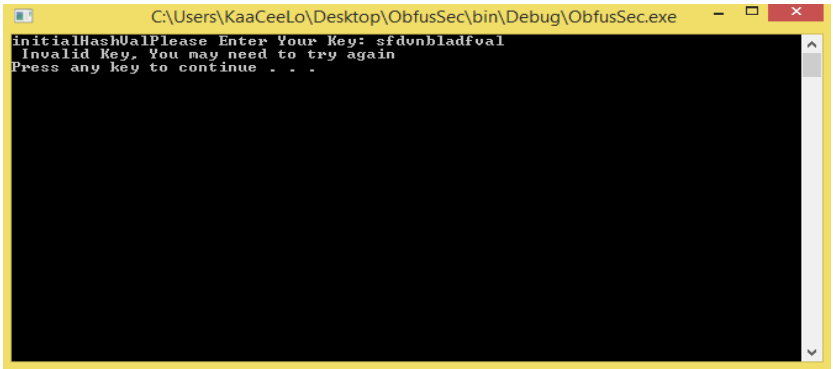

Fig. 7: Screen display showing the text string "Invalid Key, please try again"

In Figure 7, we opened our executable file, created by running our C++ source code, in the Hacker's Disassembler. We then searched for the text string "Invalid Key, please try again". We discovered that the text string cannot be found in the executable file. The reason for this is because the serial number authentication scheme has been encrypted and obfuscated. So the text string can never be seen by the cracking tool. This means that the executable file cannot be cracked since every cracking process requires that you should search for the displayed text strings after entering the wrong serials before cracking can take place. Without detecting the text string generated by entering the wrong key, we cannot know the offset address of the jump call required for modification of the executable to take place. This means our hybridized self - modifying mechanism cannot be cracked. Thus the wrong key is typed and the text strings "Invalid key, you may need to try again" is shown on the screen display. This text string is very important to us because we need it to commence our cracking processes. We have to search for the text string " Invalid key, you may need to try again " using the search button at the top of the dialogue box in the Hacker's Disassembler. If we can locate the text string in the hacker's disassembler, then we can proceed with the cracking processes.

V. CONCLUSION

Security is a very crucial aspect of computer science and it plays a major role in the advancement of Information Technology especially with the growing use of computer software for e – commerce, e – banking and for governance in all areas of life where national security is a very critical issue. It is therefore relevant to study this area with increased efforts so that software produced and made - available for the general public can be more secured and reduce the possibilities of being attacked by crackers or hackers. In this paper, we used source code obfuscation technique to checkmate software reverse engineering which is one of the techniques hackers often used to crack software. Using this technique, the original code is mangled such that the resultant code, the obfuscated code, is difficult to analyze by the hackers, although; the code retains its originality. We discovered that the text string cannot be found in the executable file. The reason for this is because the serial number authentication scheme has been encrypted and obfuscated. So the text string can never be seen by the cracking tool. This means that the executable file cannot be cracked since every cracking process requires that you should search for the displayed text strings after entering the wrong serials before cracking can take place. Without detecting the text string generated by entering the wrong key, we cannot know the offset address of the jump call required for modification of the executable to take place. This means our hybridized self - modifying mechanism cannot be cracked.

REFERENCES

[1] Boldewin, F. (2012). The Big SoftICEHow-to. http://www.reconstructer.org/papers/The%20big%20SoftICE% 20howto.pdf (consulted on February 10, 2012).

[2] Chikofsky, E. and Cross, J. (1990). Reverse Engineering and Design Recovery: A Taxonomy', IEEE Software, Vol. 7, pp. 13–17.

[3] Cifuentes, C., and Gough, K. (1995). Decompiling Of Binary Programs. Software – Practice & Experience, Vol. 25, No 7, pp. 811–829.

[4] Collberg, C. and Thomborson, C. (2002). Watermarking, Tamper-Proofing, And Obfuscation - Tools for Software Protection, IEEE Transactions on Software Engineering, Vol. 28, Issue: 8, pp. 735 – 746.

[5] Data Rescue. IDA Pro (2012), http://www.datarescue.com/idabase/ Accessed February 10, 2012.

[6] Gopal, R. and Snaders, G. (1998). International Software Piracy: Analysis of Key Issues and Impacts. Info. Sys. Research, Vol. 9, No. 4, pp. 380-397.

[7] Jain, A., Jason, K., Jordan, S. and Brian, T. (2007). Software Cracking, http://courses.ece.ubc.ca/412/previous_years/2007_1_spring/modules/term_project/reports/2007/software_cracking.pdf, Accessed August 5, 2013.

[8] Jakobsson, M. and Reiter, M. (2002). Discouraging Software Piracy Using Software Aging, Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer LNCS 2320, pp.1–12.

[9] Kammerstetter, M., Platzer, C. and Wondracek, G. (2012). Vanity, Cracks and Malware: Insights into the Anti-Copy Protection Ecosystem, Proceedings of The 2012 ACM Conference On Computer And Communications Security, pp. 809-820.

[10] Kini, R., Rominger, A. and Vijayaraman, B. (2000). An Empirical Study of Software Piracy and Moral Intensity Among University Students. The Journal of Computer Information Systems, Vol. 40, pp. 62-72.

[11] Krügel, C., Robertson, W., Valeur, F. and Vigna, G. (2004). Static Disassembly of Obfuscated Binaries. In USENIX Security Symposium, pp. 255–270.

[12] Linn, C. and Debray, S. (2003). Obfuscation of Executable Code To Improve Resistance To Static Disassembly. In S. Jajodia, V. Atluri, And T. Jaeger, Editors, ACM Conference on Computer and Communications Security, pp. 290–299.
[13] Madou, M., Anckaert, B., Moseley, P., Debray, S., De Sutter, B. and De Bosschere, K. (2005). Software Protection Through Dynamic Code Mutation, WISA '05 Proceedings of the 6th international conference on Information Security Applications, pp. 194-206.
[14] Madou, M., Anckaert, B., De Sutter, B. and De Bosschere, K. (2005). Hybrid Static dynamic Attacks Against Software Protection Mechanisms. In R. Safavi-Naini And M. Yung, Editors, Digital Rights Management Workshop, pp. 75–82.
[15] Proebsting, T. and Watterson, S. (1997). Krakatoa: Decompilation In Java (Does Bytecode Reveal Source?). In COOTS, pp. 185–198.USENIX.
[16] Schwarz, B., Debray, S. and Andrews, G. (2002). Disassembly Of Executable Code Revisited. In A. Van Deursenand E. Burd, Editors, WCRE, IEEE Computer Society, pp. 45–54.
[17] Stallman, R., Pesch, R. and Shebs, S. (2010), Debugging With gdb: The GNU Source-Level Debugger.

Appendix: The Obfuscated version of the C++ code

```
// ObfusSec.cpp
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include<iostream>
#include<fstream>
#include <iomanip>
#include <sstream>
#include <windows.h>
#define O int
#define __OxCXs return
#define O_Ox__ }
#define __O_ {
#define Ax0xBEGIN void
#define _O_ Ax0xBEGIN
#define _C_ char
#define OxcC_ _C_
#define OxK__0x654hu string
#define _010011101 bool
#define oxFEABD  if
#define o0Oo while
#define o0 cout

typedef unsigned __int64 uint64_t;
using namespace std;OxK__0x654hu Ox11(ifstream&);
OxK__0x654hu _0010();
OxK__0x654hu _O__(OxK__0x654hu toEncrypt);
class HashInfo
__O_ OxcC_ hashval[100];OxcC_ configHash[100];O run_state;OxcC_ appkey[20];
public:Ax0xBEGIN setHashVal(OxK__0x654hu);OxcC_* getHashVal();Ax0xBEGIN setState(O);
O getState();Ax0xBEGIN setConfigHash(OxK__0x654hu);OxcC_* getConfigHash();
Ax0xBEGIN setAppKey(OxK__0x654hu);OxcC_* getAppKey();O_Ox__;
Ax0xBEGIN HashInfo :: setHashVal(OxK__0x654hu hashv)__O_ strncpy(hashval, hashv.c_str(),
sizeof(hashval));hashval[sizeof(hashval) - 1] = 0;}Ax0xBEGIN HashInfo :: setState(O state)
__O_ run_state = state;}O HashInfo :: getState()__O_ __OxCXs run_state;O_Ox__
OxcC_* HashInfo :: getHashVal()__O_ __OxCXs hashval;O_Ox__
Ax0xBEGIN HashInfo :: setConfigHash(OxK__0x654hu hashv)
__O_ strncpy(configHash, hashv.c_str(), sizeof(configHash));configHash[sizeof(configHash) - 1] = 0;O_Ox__
OxcC_* HashInfo :: getConfigHash()__O_ __OxCXs configHash;O_Ox__
Ax0xBEGIN HashInfo :: setAppKey(OxK__0x654hu key)__O_ strncpy(appkey, key.c_str(),
sizeof(appkey));appkey[sizeof(appkey) - 1] = 0;O_Ox__
OxcC_* HashInfo :: getAppKey()__O_ __OxCXs appkey;O_Ox__
```

```
O MAX(O x, O y)__O_  oxFEABD  ((x) > (y))__OxCXs x;else
__OxCXs y;O_Ox__
//-----------------------------------------------------------------------------------------------
_C_ ox11L[10] = __O_'\
__O_O_Ox__ 0o','0xBE-0xBD1','\
__O_O_Ox__ obfuscated2','123','\
__O_O_Ox__ 0x1234','A1b2c#d45','000000006','0xuif7','__8','_XXxXX9'
O_Ox__;O __0xEOF();Ax0xBEGIN ___$xBEGIN(O,O _O,OxcC_ _0);
_O_ _O1(O,_C_);_O_ _O1(O,OxK__0x654hu);
O main(O argc, const OxcC_ *argv[])__O_ HashInfo sobj;
_010011101 rit_new_hash = false;ifstream in("config.dat");
oxFEABD (in.is_open())__O_ o0Oo(!in.eof())__O_ in.read((OxcC_*) &sobj, sizeof(sobj));
oxFEABD (!in.eof()) __O_ OxK__0x654hu mhash; oxFEABD (sobj.getState()==0) __O_
OxK__0x654hu chash; stringstream cst;
o0 << sobj.getHashVal(); OxK__0x654hu hh(sobj.getHashVal());
oxFEABD (hh=="initialHashVal" && "1" == "1") __O_ OxK__0x654hu cmdkey;
o0 << " Please Enter Your Key: "; cin >> cmdkey;
OxK__0x654hu ckey(sobj.getAppKey());
oxFEABD (cmdkey != ckey) __O_ OxK__0x654hu tmpp = "k%=*'\"/k .2gk$>k&*2k%../k?$k?92k*,*\"%";
o0 << _O__(tmpp) <<endl;system("pause"); exit(1);
O_Ox__ else __O_ OxK__0x654hu tmpp = "k .2k\n"
"((.;?./"; o0 << _O__(tmpp) <<endl;
system("pause"); O_Ox__ ifstream f2; f2.open(argv[0], ios::in | ios::binary | ios::ate);
oxFEABD  (!f2.is_open()) __O_ cerr << " Error opening main executable file" << endl;
system("pause");exit(1);O_Ox__ mhash = Ox11(f2);f2.close();
OxK__0x654hu new_computed_hash = mhash;sobj.setHashVal(new_computed_hash);
sobj.setState(1);o0 << " This is the first time: New Hash Has Been Set \n";
rit_new_hash = true;O_Ox__ else __O_
OxK__0x654hu tmpp = "k%=*\"/k*\"/*?\"$%k$%k-\"98?k9>%ek'\" .'2k($%-\",k-\"'.k&>?\"'*?\"$%";
o0 << _O__(tmpp) <<endl;exit(0);O_Ox__ O_Ox__ else  oxFEABD (sobj.getState()==1)
__O_ ifstream f2;f2.open(argv[0], ios::in | ios::binary | ios::ate);
oxFEABD  (!f2.is_open()) __O_ cerr << " Error opening main executable file 2 \n" << endl;
system("pause"); exit(1);O_Ox__ mhash = Ox11(f2);f2.close();
OxK__0x654hu new_computed_hash = mhash;OxK__0x654hu hh(sobj.getHashVal());
oxFEABD (hh != new_computed_hash) __O_ OxK__0x654hu tmpp = "k%=*'\"/k\n"
">?#.%?\"(*?\"$%k$%k?#\"8k&*(#\"%.";o0 << _O__(tmpp) << endl;
system("pause");exit(1);O_Ox__ else __O_ o0 << " Application Initialization Successful \n ";
O_Ox__ O_Ox__ o0 << sobj.getHashVal();O_Ox__ O_Ox__ O_Ox__ in.close();
oxFEABD (rit_new_hash) __O_ ofstream out; out.open("new.dat", ios::out| ios::binary);
ifstream in2("config.dat"); oxFEABD (!in2.is_open()) __O_
cerr << " Config file not found err:3 \n";system("pause");exit(0);O_Ox__
else __O_ out.write((OxcC_*) & sobj, sizeof(sobj));
O_Ox__ out.close(); in2.close();remove("config.dat");
rename("new.dat","config.dat");O_Ox__ system("pause");
O player = 1,i,choice;_C_ mark;do __O_ ___$xBEGIN(0xE0F > 0 ? 0 : 2 , 0xFF , 0xAB);
player=(player%2)?1:2; o0 << "Player " << player << ", enter a number:  ";cin >> choice;
mark=(player == 1) ? 'X' : 'O'; oxFEABD  (choice == 1 && ox11L[1] == '1')
ox11L[1] = mark; else  oxFEABD  (choice == 2 && ox11L[2] == '2') ox11L[2] = mark;
else  oxFEABD  (choice == 3 && ox11L[3] == '3')
ox11L[3] = mark;else  oxFEABD  (choice == 4 && ox11L[4] == '4')ox11L[4] = mark;
else  oxFEABD  (choice == 5 && ox11L[5] == '5')
ox11L[5] = mark;else  oxFEABD  (choice == 6 && ox11L[6] == '6')
ox11L[6] = mark;else  oxFEABD  (choice == 7 && ox11L[7] == '7')
ox11L[7] = mark; else  oxFEABD  (choice == 8 && ox11L[8] == '8')
ox11L[8] = mark; else  oxFEABD  (choice == 9 && ox11L[9] == '9')
ox11L[9] = mark; else __O_ o0<<"Invalid move ";player--;cin.ignore();cin.get();
O_Ox__ i=__0xEOF();player++;O_Ox__ o0Oo(i==-1); ___$xBEGIN(0xE0F>2?1:2,2,'B');
oxFEABD (i==1?true:false) o0<<"==>\aPlayer "<<--player<<" win ";
else o0<<"==>\aGame draw";cin.ignore();cin.get();system("pause");__OxCXs 0;
```

```
O_Ox__ O __0xEOF() __O_ oxFEABD  (ox11L[1] == ox11L[2] && ox11L[2] == ox11L[3])
__OxCXs 1; else  oxFEABD  (ox11L[4] == ox11L[5] && ox11L[5] == ox11L[6]) __OxCXs 1;
else  oxFEABD  (ox11L[7] == ox11L[8] && ox11L[8] == ox11L[9])__OxCXs 1;
else  oxFEABD  (ox11L[1] == ox11L[4] && ox11L[4] == ox11L[7])__OxCXs 1;
else  oxFEABD  (ox11L[2] == ox11L[5] && ox11L[5] == ox11L[8])__OxCXs 1;
else  oxFEABD  (ox11L[3] == ox11L[6] && ox11L[6] == ox11L[9])__OxCXs 1;
else  oxFEABD  (ox11L[1] == ox11L[5] && ox11L[5] == ox11L[9])
__OxCXs 1;else  oxFEABD  (ox11L[3] == ox11L[5] && ox11L[5] == ox11L[7])
__OxCXs 1;else  oxFEABD  (ox11L[1] != '1' && ox11L[2] != '2' && ox11L[3] != '3'
&& ox11L[4] != '4' && ox11L[5] != '5' && ox11L[6] != '6'
&& ox11L[7] != '7' && ox11L[8] != '8' && ox11L[9] != '9') __OxCXs 0; else
__OxCXs -1;O_Ox__ Ax0xBEGIN ___$xBEGIN(O,O _O,OxcC_ _0)__O_
system("cls");o0 << "\n\n\tTic Tac Toe\n\n";
o0 << "Player 1 (X)  -  Player 2 (O)" << endl << endl;o0 << endl;
_O1(5,' ');_O1(1,'|');_O1(5,' ');_O1(1,'|');_O1(5,' ');
_O1(1,'\n');o0 << "  " << ox11L[0xE0-0xDF] << "  |  " << ox11L[0xDF-0xDD] << "  |  " << ox11L[0x4D-0x4A]
<< endl;_O1(5,'_');
_O1(1,'|');_O1(5,'_');_O1(1,'|');_O1(5,'_');_O1(1,'\n');_O1(5,' ');_O1(1,'|');_O1(5,' ');_O1(1,'|');_O1(5,' ');
                        _O1(1,'\n');_O1(2,' ');
                    o0 << ox11L[-0xFA+0xFE];_O1(2,' ');
               _O1(1,'|');_O1(2,' ');o0 << ox11L[-0xCB+0xD0];
          _O1(2,' ');_O1(1,'|');_O1(2,' ');o0 << ox11L[0xD0-0xCA] << endl;
         _O1(5,'_');                                    _O1(1,'|');
          _O1(5,'_');                                  _O1(1,'|');
          _O1(5,'_');_O1(1,'\n');_O1(5,' ');_O1(1,'|');_O1(5,' ');
                _O1(1,'|');_O1(5,' ');_O1(1,'\n');_O1(2,' ');
                    o0 << ox11L[0x5C-(0x2B+0x2A)];
                       _O1(2,' ');_O1(1,'|');
_O1(2,' ');o0 << ox11L[0x2B-0x23];_O1(2,' ');_O1(1,'|');_O1(2,' ');
o0 << ox11L[0xAA-0xA1] << endl;_O1(5,' ');_O1(1,'|');_O1(5,' ');
_O1(1,'|');_O1(5,' ');_O1(1,'\n');O_Ox__ _O_ _O1(O __o,_C_ o__)
__O_ for(O _o=0; _o<__o; _o++) __O_ printf ("%c",o__);O_Ox__
O_Ox__ OxK__0x654hu Ox11(ifstream& f)__O_ uint64_t hash, fsize;f.seekg(0, ios::end);fsize = f.tellg();
f.seekg(0, ios::beg);hash = fsize;for (uint64_t tmp = 0, i = 0; i < 65536 / sizeof (tmp)
&& f.read((OxcC_*) &tmp, sizeof (tmp)); i++, hash += tmp);f.seekg(MAX(0, (uint64_t) fsize - 65536),
ios::beg);
for (uint64_t tmp = 0, i = 0; i < 65536 / sizeof (tmp) && f.read((OxcC_*) &tmp, sizeof (tmp)); i++, hash +=
tmp);
stringstream cst;cst << setw(16) << setfill('0') << hex << hash;__OxCXs cst.str()+_0010();
O_Ox__ OxK__0x654hu _0010() __O_ HW_PROFILE_INFO hwProfileInfo;
oxFEABD  (GetCurrentHwProfileA(&hwProfileInfo) != 0) __O_ OxK__0x654hu
_10011 = hwProfileInfo.szHwProfileGuid;
_10011.erase(0, 1); _10011 = _10011.substr(0, _10011.size() - 1);
__OxCXs _10011; O_Ox__ else __O_ cerr << " Can not get system unique id";
exit(0); O_Ox__ O_Ox__ OxK__0x654hu _O__(OxK__0x654hu Ox)
__O_ OxcC_ key = 'K';OxK__0x654hu output = Ox;
for (O i = 0; i < Ox.size(); i++)output[i] = Ox[i] ^ key; __OxCXs output;O_Ox__
```