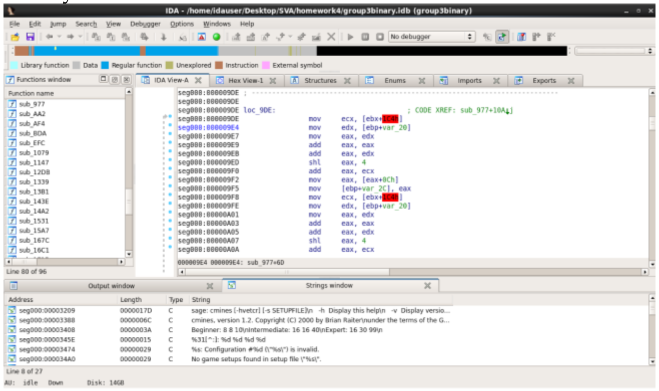
# Reverse Engineering Executive Document for cmines binaries(both modified and authentic) using IDA-Pro disassembler (revised November 2018)

Jay Chow, Kevin Hamilton, and Yuqing Wang, MSSI graduate students, JHUISI

**Abstract** - We reverse-engineered a modified, incomplete binary that we initially had no clue about. Expectedly, it had certain information stripped off such as symbols but still contained important information that led us to figure out the actual gaming application that was open-sourced. It turned out to be cgames, a 32-bit linux binary. Then, we ran IDA-Pro on both the modified binary and authentic binary to uncover certain information by analyzing the dis-assembled x-86 assembly language. This document highlights the usage of IDA-Pro and our interactions with this powerful software

## 1 Project Goals

Our goal as a group was to obtain a hands-on practice using the IDA-Pro application with two different binaries, a modified binary with certain information stripped off from the executable such as symbols, and the authentic binary which we were able to git clone from https://github.com/BR903/cgames.git. We obtained a hands-on practice using the IDA-Pro application on both executables. When we initially loaded the modified binary into IDA-Pro:



```
IDA - /home/idauser/Desktop/SVA/homework4/group3binary.idb (group3binary)

seg000:000009DE ; ---------------------------------------------------------------
seg000:000009DE
seg000:000009DE loc_9DE:                                ; CODE XREF: sub_977+10A↓j
seg000:000009DE                 mov     ecx, [ebx+1C4h]
seg000:000009E4                 mov     edx, [ebp+var_20]
seg000:000009E7                 mov     eax, edx
seg000:000009E9                 add     eax, eax
seg000:000009EB                 add     eax, edx
seg000:000009ED                 shl     eax, 4
seg000:000009F0                 add     eax, ecx
seg000:000009F2                 mov     eax, [eax+0Ch]
seg000:000009F5                 mov     [ebp+var_2C], eax
seg000:000009F8                 mov     ecx, [ebx+1C4h]
seg000:000009FE                 mov     edx, [ebp+var_20]
seg000:00000A01                 mov     eax, edx
seg000:00000A03                 add     eax, eax
seg000:00000A05                 add     eax, edx
seg000:00000A07                 shl     eax, 4
seg000:00000A0A                 add     eax, ecx
000009E4 000009E4: sub_977+6D
```

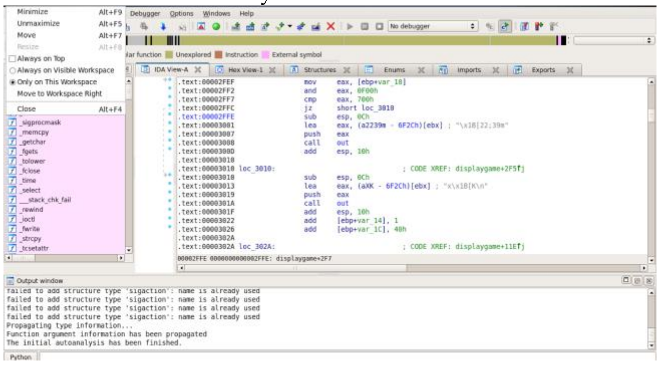
```
Function name
sub_977
sub_AA2
sub_AF4
sub_BDA
sub_EFC
sub_1079
sub_1147
sub_12DB
sub_1339
sub_13B1
sub_143E
sub_14A2
sub_1531
sub_15A7
sub_167C
sub_16C1
Line 80 of 96
```

```
Address              Length    Type String
seg000:00003209      0000017D  C    sage: cmines [-hvetcr] [-s SETUPFILE]\n  -h  Display this help\n  -v  Display versio...
seg000:00003388      0000006C  C    cmines, version 1.2. Copyright (C) 2000 by Brian Raiter\nunder the terms of the G...
seg000:00003408      0000003A  C    Beginner: 8 8 10\nIntermediate: 16 16 40\nExpert: 16 30 99\n
seg000:0000345E      00000015  C    %31[^:]: %d %d %d %d
seg000:00003474      00000029  C    %s: Configuration #%d (\"%s\") is invalid.
seg000:000034A0      00000029  C    No game setups found in setup file \"%s\".
Line 8 of 27
AU:  idle   Down        Disk: 14GB
```

**Fig. 1. IDA-Pro view of group3binary, a modified binary from the original cmines executable**

### 1.1 Overview

IDA-Pro is a disassembler that supports x86 architecture and several file formats such as PE and COFF, ELF and a.out. IDA-Pro dissembles a binary in machine code to perform function discovery, stack analysis and local variable identification. When an executable loads into IDA-Pro, the program maps the executable into memory.



```
.text:00002FEF                 mov     eax, [ebp+var_18]
.text:00002FF2                 and     eax, 0F00h
.text:00002FF7                 cmp     eax, 700h
.text:00002FFC                 jz      short loc_3010
.text:00002FFE                 sub     esp, 0Ch
.text:00003001                 lea     eax, (a2239m - 6F2Ch)[ebx] ; "\x1B[22;39m"
.text:00003007                 push    eax
.text:00003008                 call    out
.text:0000300D                 add     esp, 10h
.text:00003010
.text:00003010 loc_3010:                               ; CODE XREF: displaygame+2F5↑j
.text:00003010                 sub     esp, 0Ch
.text:00003013                 lea     eax, (aXK - 6F2Ch)[ebx] ; "x\x1B[K\n"
.text:00003019                 push    eax
.text:0000301A                 call    out
.text:0000301F                 add     esp, 10h
.text:00003022                 add     [ebp+var_14], 1
.text:00003026                 add     [ebp+var_1C], 40h
.text:0000302A
.text:0000302A loc_302A:                               ; CODE XREF: displaygame+11E↑j
00002FFE 0000000000002FFE: displaygame+2F7
```

```
_sigprocmask
_memcpy
_getchar
_fgets
_tolower
_fclose
_time
_select
___stack_chk_fail
_rewind
_ioctl
_fwrite
_strcpy
_tcsetattr
```

```
failed to add structure type 'sigaction': name is already used
failed to add structure type 'sigaction': name is already used
failed to add structure type 'sigaction': name is already used
failed to add structure type 'sigaction': name is already used
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
```

**Fig. 2. IDA-Pro view of cmines, a 32-bit game that runs on a 64-bit Linux system**

IDA-Pro is useful for binary programs that source code is unavailable. More importantly, it shows users assembly that is executed by the CPU. We were also exposed to IDA-Pro being used as a debugger, an interactive tool and programmable using plugins. Additionally, IDA-Pro is beneficial for malware analysis to investigate software viruses utilized by anti-virus companies, security research companies, software development companies, agencies and military organizations.

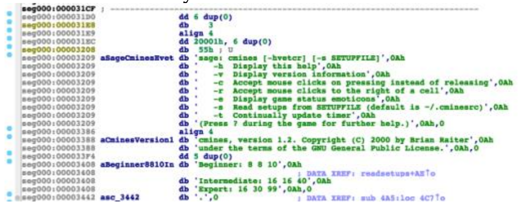
## 2 Project execution

Our initial analysis was done on a unnamed binary. We were able to determine that this file was a ELF executable by two features in the code. The first was the fact that IDA was able to parse assembly instructions. The data section contained strings such as .bss, .init, .fini, and .plt. From our experience with the ELF format earlier in the semester those are distinctive features of the ELF format. The ELF file was stripped on function and variable names. The challenge of this assignment was to determine certain functions and variables by the assembly instructions near their context. Our first step was to look for predefined strings and where they were used. This gave us a hint at the functionality of certain subroutines. We used the rename ability in IDA which made it easier for our team to see interdepecies between functions. We also examined the binary as hex and confirmed that the file did not contain the ELF's magic number in the first bytes. We believe that the header of file was removed from the executable.

## 3 Accomplishments

We learned many things about IDA-Pro from reading books and online references. For example, a key accomplishment is that we figured the modified binary called group3binary was stripped of symbols. As such, IDA-Pro's recursive descent algorithm was unable to locate a call to the main() function in the modified binary. We gained a clear understanding that sub_xxx is a label that IDA uses for functions identified by the recursive descent algorithm and that loc_xxx is a label that IDA uses for branches within a function.

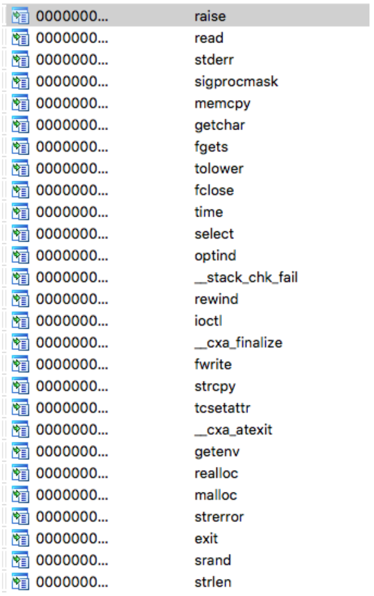
We use IDA-Pro to identify the binary and successfully figure out that it is a 32-bit cmines game for Linux. The initialized string values gave us a hint about what the authentic binary was.



```
seg000:000031CF ; ---------------------------------------------------------------
seg000:000031D0                 dd 6 dup(0)
seg000:000031E8                 db    3
seg000:000031E9                 align 4
seg000:000031EC                 dd 20001h, 6 dup(0)
seg000:00003208                 db  55h ; U
seg000:00003209 aSageCminesHvet db 'sage: cmines [-hvetcr] [-s SETUPFILE]',0Ah
seg000:00003209                 db '    -h  Display this help',0Ah
seg000:00003209                 db '    -v  Display version information',0Ah
seg000:00003209                 db '    -c  Accept mouse clicks on pressing instead of releasing',0Ah
seg000:00003209                 db '    -r  Accept mouse clicks to the right of a cell',0Ah
seg000:00003209                 db '    -e  Display game status emoticons',0Ah
seg000:00003209                 db '    -s  Read setups from SETUPFILE (default is ~/.cminesrc)',0Ah
seg000:00003209                 db '    -t  Continually update timer',0Ah
seg000:00003209                 db '(Press ? during the game for further help.)',0Ah,0
seg000:00003386                 align 4
seg000:00003388 aCminesVersion1 db 'cmines, version 1.2. Copyright (C) 2000 by Brian Raiter',0Ah
seg000:00003388                 db 'under the terms of the GNU General Public License.',0Ah,0
seg000:000033F4                 dd 5 dup(0)
seg000:00003408 aBeginner8810In db 'Beginner: 8 8 10',0Ah
seg000:00003408                                        ; DATA XREF: readsetups+AE↑o
seg000:00003408                 db 'Intermediate: 16 16 40',0Ah
seg000:00003408                 db 'Expert: 16 30 99',0Ah,0
seg000:00003442 asc_3442        db '.',0               ; DATA XREF: sub_4A5:loc_4C7↑o
```

**Fig. 1. String values in assembly**

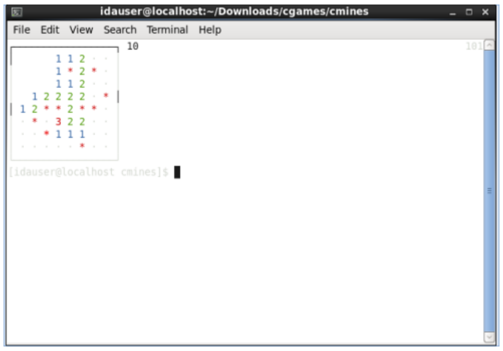
Then we followed the string calls and control flows to locate the function we need. We successfully found the readsetups function by using the string signatures and record the offset address. Although we found some assembly parts which can be the main function quickly, it still took us some time to verify our answer. Besides the readsetups, we identified some other interesting functions in the assembly based on the string signatures. To make the assembly more readable, we named the functions we found which include readsetups, init and memeoryCheck. We also checked the function window to list the imported functions used by the binary.



| | |
|---|---|
| 0000000... | raise |
| 0000000... | read |
| 0000000... | stderr |
| 0000000... | sigprocmask |
| 0000000... | memcpy |
| 0000000... | getchar |
| 0000000... | fgets |
| 0000000... | tolower |
| 0000000... | fclose |
| 0000000... | time |
| 0000000... | select |
| 0000000... | optind |
| 0000000... | __stack_chk_fail |
| 0000000... | rewind |
| 0000000... | ioctl |
| 0000000... | __cxa_finalize |
| 0000000... | fwrite |
| 0000000... | strcpy |
| 0000000... | tcsetattr |
| 0000000... | __cxa_atexit |
| 0000000... | getenv |
| 0000000... | realloc |
| 0000000... | malloc |
| 0000000... | strerror |
| 0000000... | exit |
| 0000000... | srand |
| 0000000... | strlen |

**Fig. 2. Imported functions used by cmine**

We recreate a 32-bit version of cmine and run it on our Ubuntu 18.04 platform. It is necessary to edit the makefile before we build it. There are also some libraries to link before the build.



idauser@localhost:~/Downloads/cgames/cmines
File Edit View Search Terminal Help

```
[idauser@localhost cmines]$ 
```

**Fig. 3. 32-bit version Cmines game**

Since we already have the source code now, we can check whether we found the correct function by comparing the assembly. It seems like we got the right one. Then we moved on with locating the specific variable "buf". We noticed that there is a buffer which is defined in the function readsetups(). We checked it in the ida view and found that it is an array of 256 bytes.

We also compared the version with without debugging symbols with the original incomplete one to find out the difference. Finally, we build a version with -O3 and compare it with the original one. It seems like this version has less functions than the before. The functions nearly the same size with the missing parts, which mean the setting -O3 optimize these parts in the compiling process.

## 4 Lessons learned

We learned a great deal about IDA-Pro through reading “Practical Malware Analysis”, online references and conducting this lab:

1) The Binary File option when loading an executable can be useful as it could check appended shellcode that was inserted, additional data and encryption information that are parts of malware.
2) There are 2 modes in the disassembly window, which are the graph mode that shows the control flow during analysis. Green color indicates if a conditional jump is taken. Red color means otherwise.
3) Several useful windows for analysis are provided by IDA. For example, you can list all the functions in the executable, list every address which can be a function, code, data or string with a name in the name window, show all the strings in string window, all imports in the imports window, list all exported functions in exports window and list the layout of all active data structures in structure windows.
4) Click the Text tab in Search for searching a specific string in the whole disassembly file.

IDA Pro recognizes features, marks functions and decomposes the local variables. When we used the authentic executable for static analysis in IDA-Pro, IDA could locate local variables and parameters in the function. It can label local variables with var_ as the prefix and parameters with arg_ as the prefix, and the suffix corresponding to their offset to $EBP. We figured that local variables, relative to $EBP, are at negative offsets while arguments, relative to $EBP, are at positive offsets. We also appreciated the fact that IDA-Pro allows cross-reference graphs for both the entire program and a single function.

Lastly, We learned to modify the disassembly to suit our needs. For example, we can rename variable and function names to be more easily understood. In addition, we could also embed comments. We also learned something useful to locate the functions and variables we need in IDA-Pro. For example, we can use different views in IDA to find the assembly we want. We can follow the control flow to find where the functions or variables are called.

## 5 References

[1] Michael Sikorski and Andrew Honig, “Practical Malware Analysis”, The Hands-On Guide to Dissecting Malicious Software

[2] Copyright 2009 Hex-Rays SA, https://www.hex-rays.com/products/ida/ida-executive.pdf