Reversing to Detect Software Vulnerabilities

Bruno Gonçalves de Oliveira, Maycon Vitali
Trustwave
Sao Paulo, SP - Brazil

Dr. Willian Massami Watanabe
UTFPR
Cornelio Procopio, PR - Brazil

Abstract — **Security is a real concern for the society and it is not different for software. Vulnerable applications can expose the users to multiple risks. Software disassembling is an interesting approach to discover vulnerabilities. Unmounting an application into Assembly code can reveal multiple characteristics from the software. During the disassembling, technical features can be revealed that enable the researcher to discover issues even with no source-code available. The paper has the goal of demonstrating the disassembling efficacy for detecting vulnerabilities on software and presents a case study to evaluate the concept.**

Keywords - Software; Reverse engineering; Disassembling; Vulnerabilities; Security.

I. INTRODUCTION

A mature secure software development process consists basically in applying techniques to prevent and identify security flaws on every phase of the development. It is useful to identify vulnerabilities since the design phase until after the implementation process. In the design phase, a very useful tool is the threat modelling, where it is possible to identify design vulnerabilities that would be expensive to correct after the final release. During the code writing process, code review techniques can be applied with the goal of identifying security issues that are easy to detect with the code itself or static analysis tools. Once the software is finished, a vulnerability assessment can be useful to detect possible vulnerabilities that can be hard to identify during a code review process.

The final assessment of the development process is conducted through software security tests which are helpful for developers and also for the users' community, since the results can benefit everybody that is engaged to the software. For example, identifying issues that can turns the application vulnerable to attacks that compromise data integrity, confidentiality or the availability. Considering specifically the Internet Explorer, a worldwide software, a vulnerability with no correction, also known as 0-day vulnerability, can possibly impact millions of users[1]. In most serious cases, attackers would be able to execute arbitrary remote commands and have full access of the systems. Only at the website Exploit Database [2], there are 405 public tools, exploits, for compromising vulnerabilities on Internet Explorer.

Nowadays, software is becoming more and more complex and it is very expensive to assess its security. The source-code is not always accessible since many applications are copyrighted and the source-code distribution is not authorized. That becomes another obstruction for the source-code review assessment.

The software reverse engineer technique does not require the complete source-code for the assessment, because disassembling and debugging methods can transcribe the binary code to a human readable code, Assembly language, allowing the understanding. Utilizing the Assembly, it is possible to observe the functions that can trigger vulnerabilities in the system.

It's important to note that, in some cases, the software is written in a high-level language such as Java or C#, which compiles the source-code to an intermediate language called byte-codes. Byte-code is interpreted by a virtual machine (JDK for Java or .NET Framework for C#), and it can be easily translated to the original source-code, making unnecessary to read the low-level Assembly code.

The article introduces how software reversing technique, more specifically disassembling, can be useful for detecting vulnerabilities. This work has the goal of demonstrating this technique and also the most common security problems that can be identified utilizing it.

In the case study reported in this paper, it will be possible to observe the technical efficiency for identifying security issues on the Assembly code.

This paper is organized as follow: Section II introduces the Reverse Engineering and how useful tools can be used to retrieve sensitive data from binary application; Section III is presenting the Disassembling technique; Section IV is dedicated to a case study; Section V for related work and on Section VI presents the final remarks, a conclusion of the article including future works.

II. REVERSE ENGINEERING

Reverse engineering technique is the action of reverting the steps from the development to reveal how the final product was constructed [1]. For example, an unmounted wood table will expose multiple wood pieces, bolts, and others. From this data, it is possible to replicate its construction using the same artifacts. And that helps to identify weak points from the construction to build a better one.

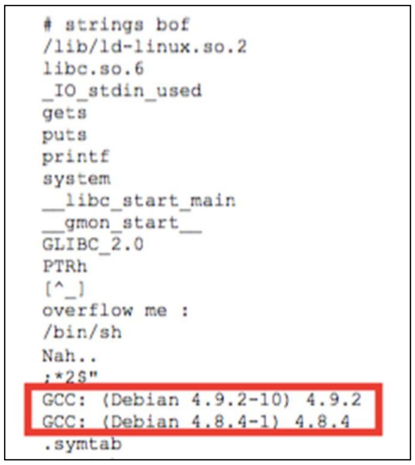
For software, it is possible to use the same concept and "unmount" the applications in order to conceive how it was developed. Binary files can be unmounted and reveal the functions that build the entire application. There are tools for working with each type of binary and architecture like *Portable*

[1] http://www.w3schools.com/browsers/browsers_stats.asp
[2] http://www.exploit-db.com/

Executable (Windows), *Executable and Linkable Format* (Unix and variants), and others [2].

To unmount an application, it is recommended a software reversing tool. Frequently both disassembler and debugger are found in one single application. The debugger can analyze the software dynamically during its execution while the disassembler does it statically, only reversing the binary to Assembly code. During the binary disassembling, information is observed that could help the understanding of application internals functionalities and also identifying vulnerabilities. For example, a way to identify a vulnerability is looking for well-known functions that usually trigger common faults like buffer overflows [3].

After disassembling the application, it is possible to detect some useful information just looking for its internal content. For example, on Figure 1, it is possible to identify the development language of the software, a hacking challenge retrieved from the website Pwnable.kr[3], only checking for *strings* on the binary. This kind of information can be very helpful for detecting security issues [4].



```
# strings bof
/lib/ld-linux.so.2
libc.so.6
_IO_stdin_used
gets
puts
printf
system
__libc_start_main
__gmon_start__
GLIBC_2.0
PTRh
[^_]
overflow me :
/bin/sh
Nah..
;*2$"
GCC: (Debian 4.9.2-10) 4.9.2
GCC: (Debian 4.8.4-1) 4.8.4
.symtab
```

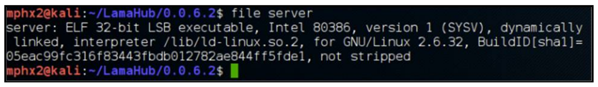
Figure 1. *Strings* from Pwnable.kr's bof.

III. Disassembling

A very important step during a software reverse engineering is the disassembling. The reverse engineering itself is a complex concept and involves much more information. This paper addresses the use of the disassembling techniques for identifying vulnerabilities. It is important to note that there are some free and commercial tools that aim to protect the software against reverse engineering techniques. Although the methods to bypass these protections are well known, the protection techniques and defeating methods will not be covered in this paper.

There are multiple tools that can disassemble software. But first, it is necessary to get some information from the file. The two main data to be retrieved are "what type of file it is" and "what architecture it is compiled". A simple "file", as seen on Figure 2, a UNIX command can get this data and help to start understanding the binary file.

[3] http://pwnable.kr/



```
mphx2@kali:~/LamaHub/0.0.6.2$ file server
server: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
 linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=
05eac99fc316f83443fbdb012782ae844ff5fde1, not stripped
mphx2@kali:~/LamaHub/0.0.6.2$
```

Figure 2. Using the UNIX command "file".

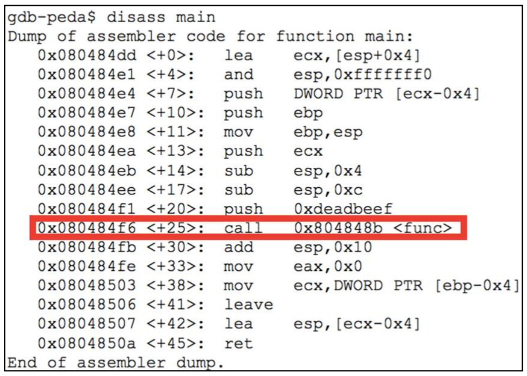
For Unix ELF files, there are debuggers/disassemblers such as The GNU Debugger (GDB), IDA Pro, Hooper and Radare2. For Windows PE binaries, there are the WinDbg, IDA Pro, OllyDBG, Immunity Debbuger and others.

These tools are very important for conducting this type of analysis, since they interpret the binary differently and frequently find different paths to unmount a file.

After disassembling, with the Assembly code, it is possible to understand the execution flow and identify weak points that can turn into vulnerabilities on the system [5].

A. Software Vulnerabilities

Understanding a software code in Assembly allows the code comprehension even in low-level with no data abstraction. This is not a trivial process, but very helpful step to discover security issues and also mandatory for exploit development. On Figure 3, a GDB disassembled code is illustrated.



```
gdb-peda$ disass main
Dump of assembler code for function main:
   0x080484dd <+0>:    lea    ecx,[esp+0x4]
   0x080484e1 <+4>:    and    esp,0xfffffff0
   0x080484e4 <+7>:    push   DWORD PTR [ecx-0x4]
   0x080484e7 <+10>:   push   ebp
   0x080484e8 <+11>:   mov    ebp,esp
   0x080484ea <+13>:   push   ecx
   0x080484eb <+14>:   sub    esp,0x4
   0x080484ee <+17>:   sub    esp,0xc
   0x080484f1 <+20>:   push   0xdeadbeef
   0x080484f6 <+25>:   call   0x804848b <func>
   0x080484fb <+30>:   add    esp,0x10
   0x080484fe <+33>:   mov    eax,0x0
   0x08048503 <+38>:   mov    ecx,DWORD PTR [ebp-0x4]
   0x08048506 <+41>:   leave
   0x08048507 <+42>:   lea    esp,[ecx-0x4]
   0x0804850a <+45>:   ret
End of assembler dump.
```

Figure 3. Pwnable.kr's bof disassembled using GDB.

There are basically three common types of security issues in software development: when unsafe functions are utilized or when safe functions are badly used. These flaws lead to memory corruption problems and depending on the severity, it is possible to control the application's execution flow through it. This problem gives the ability to execute anything else loaded on memory including malicious code. Due to the vulnerability's nature, it is only possible to work with it with a debugger to monitor the execution, memory registers and allocations [6].

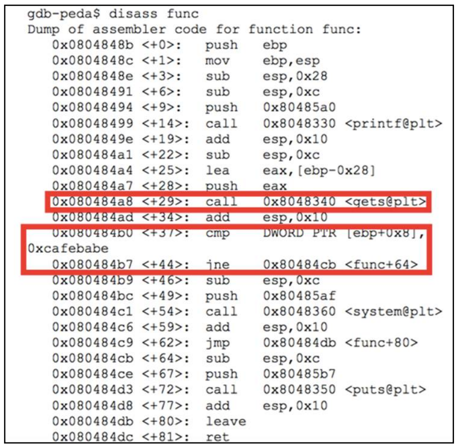
Other common vulnerability occurs when the application does not validate with signal or variable-type when performing arithmetic operations. This kind of vulnerability, usually called integer overflow, is hard to detect, especially when the affected arithmetic operations are complex and hard to understand.

On the other side, there are the business logic flaws, which imply the use of hidden or masked functions that should not be available for everyone, as for example an administration menu that can be accessed by a regular user. These issues normally are found on web applications and can be attacked with horizontal exploits [7].

Usually, it is possible to easily detect when the application uses a dangerous function, since the function name, also known as symbol name, can be retrieved. It is also possible to note, still on Figure 1, that at the application there is a reference to *gets()* function. This function does not limit the amount of data written in application variable, allowing a malicious user or process to overwrite adjacent memory.

Once disassembling the application, it is possible to note on Figure 2 that the *main()* function calls to a function called *func()*, on address 0x080484f6.

Disassembling the function *func()*, on Figure 4, there are particularities that should help understand its execution.



```
gdb-peda$ disass func
Dump of assembler code for function func:
   0x0804848b <+0>:   push   ebp
   0x0804848c <+1>:   mov    ebp,esp
   0x0804848e <+3>:   sub    esp,0x28
   0x08048491 <+6>:   sub    esp,0xc
   0x08048494 <+9>:   push   0x80485a0
   0x08048499 <+14>:  call   0x8048330 <printf@plt>
   0x0804849e <+19>:  add    esp,0x10
   0x080484a1 <+22>:  sub    esp,0xc
   0x080484a4 <+25>:  lea    eax,[ebp-0x28]
   0x080484a7 <+28>:  push   eax
   0x080484a8 <+29>:  call   0x8048340 <gets@plt>
   0x080484ad <+34>:  add    esp,0x10
   0x080484b0 <+37>:  cmp    DWORD PTR [ebp+0x8],
0xcafebabe
   0x080484b7 <+44>:  jne    0x80484cb <func+64>
   0x080484b9 <+46>:  sub    esp,0xc
   0x080484bc <+49>:  push   0x80485af
   0x080484c1 <+54>:  call   0x8048360 <system@plt>
   0x080484c6 <+59>:  add    esp,0x10
   0x080484c9 <+62>:  jmp    0x80484db <func+80>
   0x080484cb <+64>:  sub    esp,0xc
   0x080484ce <+67>:  push   0x80485b7
   0x080484d3 <+72>:  call   0x8048350 <puts@plt>
   0x080484d8 <+77>:  add    esp,0x10
   0x080484db <+80>:  leave
   0x080484dc <+81>:  ret
```

Figure 4. Disassembled *func()* function.

Still on Figure 4, it is possible to identify the function *gets()* from C language being called. As mentioned before, this function is well known for being vulnerable to buffer overflow, since there is no verification for the memory allocation from the user input [8].

In the same example, it is possible to identify the Assembly operation *cmp* at address 0x080484b0. This operation is comparing the content of the address EBP+0x8 to the value 0xcafebabe. This value could be a hidden key to something on the application. In this case, if the key is right, a shell will be executed:

```
if(key == 0xcafebabe){
system("/bin/sh");
}
```

So for executing it, it would be necessary to overwrite the address EBP + 8 bytes with the value 0xcafebabe.

There are some points that can help to identify issues on software, on the Figure 5 is shown a flowchart with some basic steps to detect vulnerabilities in Assembly codes.

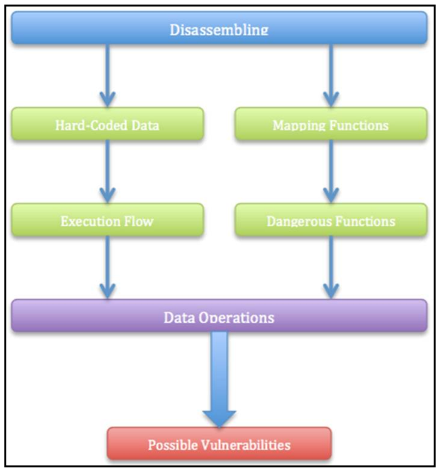


Figure 5. Disassembling to discover vulnerabilities flowchart.

From the disassembling, there are basically two paths for detecting bugs on Assembly code. It is possible to start looking at the functions list and try to identify any function that is not considered safe or can be used wrongly such as *gets, fgets, strcpy, strcmp, printf* and others [3].

After finding any possibly vulnerable function, it is necessary to analyze the data involving the function to check its exploitability. Where it is been handled, what are its arguments and how it can be triggered are some questions to be answered during the analysis to discover if the bug is a vulnerability or not.

However, if no suspicious functions are found, the researcher can start looking for hardcoded data with relevant information. Due to the language facility, any readable string can help the detection of particularities on the software and that increases its comprehension during the assessment.

Certain disassembler tools as IDA Pro, they are smart enough to recognize some characters on memory regions and mount words during the disassembling process. It's unlikely to retrieve sensitive information from hard-coded data on Assembly codes, however that still happens such as keys, passwords, etc. The readable information also will help to understand better the software flow execution.

Even with no readable variables available or at least not helpful ones, the Assembly code can be analyzed and assessed. There are dangerous operations involving data handled by the user that must be evaluated. These are possibly dangerous instructions pointed by [9] using data manipulated by an external entity, such as:

- Writing on memory, examples: MOV [ebx],edx; stos, etc.
- Conditional instruction using handled EFLAGS, such as JA, JB, JZ and others.
- Arithmetic operations, can lead to integer overflows.

From the paths to identify security issues using the disassembling, both reach the execution flow. The execution understanding is mandatory for vulnerability checking and triggering which will be necessary to prove its existence on the software.

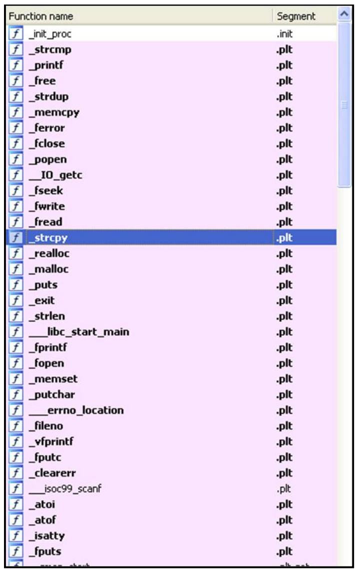
Ultimately, detecting vulnerabilities in Assembly is about understanding the data being manipulated by the operators and also locating instructions possibly vulnerable. In the next section, a case study is presented to illustrate the technique.

IV. Case Study

This paper presents a case study where the software iSQL 1.0 is security tested using previously described disassembling approach. iSQL is a command-line tool used to handle SQL syntaxes[4].

The IDA Pro was chosen for disassembling due to its capability of drawing an execution tree from the binary.

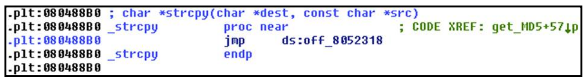
After opening the isql ELF file, it is possible to identify the functions that compose the application as illustrated on Figure 6.



Function name	Segment
_init_proc	.init
_strcmp	.plt
_printf	.plt
_free	.plt
_strdup	.plt
_memcpy	.plt
_ferror	.plt
_fclose	.plt
_popen	.plt
__IO_getc	.plt
_fseek	.plt
_fwrite	.plt
_fread	.plt
_strcpy	.plt
_realloc	.plt
_malloc	.plt
_puts	.plt
_exit	.plt
_strlen	.plt
___libc_start_main	.plt
_fprintf	.plt
_fopen	.plt
_memset	.plt
_putchar	.plt
___errno_location	.plt
_fileno	.plt
_vfprintf	.plt
_fputc	.plt
_clearerr	.plt
___isoc99_scanf	.plt
_atoi	.plt
_atof	.plt
_isatty	.plt
_fputs	.plt

Figure 6. Functions list from iSQL.

The functions provide a map on the binary. While inspecting the list, it is possible to identify a well-known vulnerable function: *strcpy()*, as detailed on Figure 7.

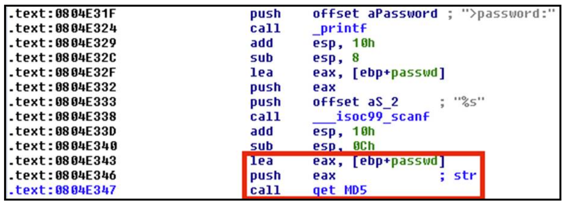


```
.plt:080488B0 ; char *strcpy(char *dest, const char *src)
.plt:080488B0 _strcpy         proc near               ; CODE XREF: get_MD5+57↓p
.plt:080488B0                 jmp     ds:off_8052318
.plt:080488B0 _strcpy         endp
.plt:080488B0
```

Figure 7. *strcpy()*.

The *strcpy()* does not validate the string size while copying, so if the destination string is not large enough for the data, it will occur a buffer overflow. The s*trcpy()* is not vulnerable by itself, the data being copied should be handled in some way by the user.

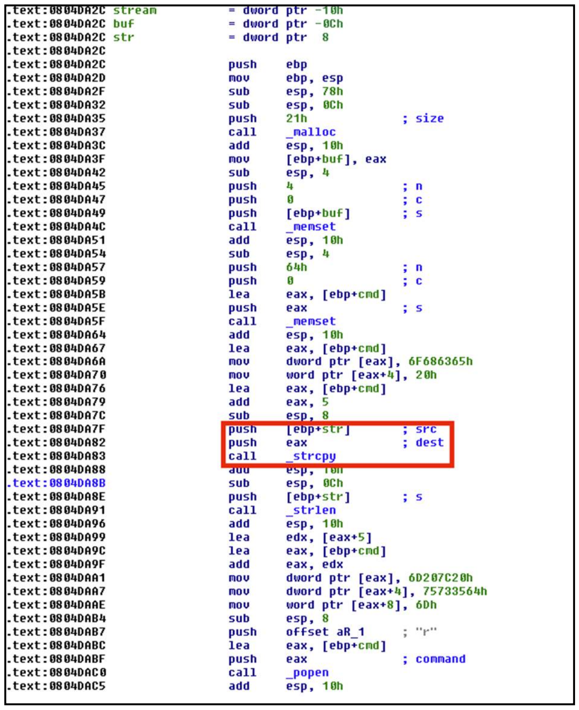
After following it, it is identified that it is been used on *get_MD5(str)* which is called by the *main()* function. The *get_MD5()* is used after the application requests the password for the database. The password is stored at EAX and then is pushed to the stack as *str*, as observed on Figure 8.



```
.text:0804E31F                 push    offset aPassword ; ">password:"
.text:0804E324                 call    _printf
.text:0804E329                 add     esp, 10h
.text:0804E32C                 sub     esp, 8
.text:0804E32F                 lea     eax, [ebp+passwd]
.text:0804E332                 push    eax
.text:0804E333                 push    offset aS_2     ; "%s"
.text:0804E338                 call    ___isoc99_scanf
.text:0804E33D                 add     esp, 10h
.text:0804E340                 sub     esp, 0Ch
.text:0804E343                 lea     eax, [ebp+passwd]
.text:0804E346                 push    eax             ; str
.text:0804E347                 call    get_MD5
```

Figure 8. *get_MD5()* call on *main()*.

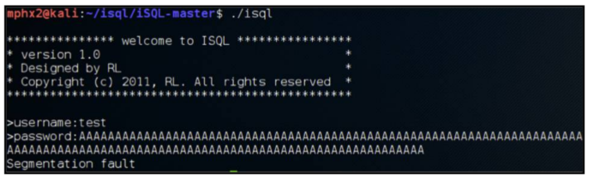
On Figure 9, there is a confirmation that *strcpy()* is using *str* as source for copying strings. Knowing that *str* is defined by the user, so this *strcpy()* function is vulnerable to buffer overflow.



```
.text:0804DA2C stream          = dword ptr -10h
.text:0804DA2C buf             = dword ptr -0Ch
.text:0804DA2C str             = dword ptr  8
.text:0804DA2C
.text:0804DA2C                 push    ebp
.text:0804DA2D                 mov     ebp, esp
.text:0804DA2F                 sub     esp, 78h
.text:0804DA32                 sub     esp, 0Ch
.text:0804DA35                 push    21h             ; size
.text:0804DA37                 call    _malloc
.text:0804DA3C                 add     esp, 10h
.text:0804DA3F                 mov     [ebp+buf], eax
.text:0804DA42                 sub     esp, 4
.text:0804DA45                 push    4               ; n
.text:0804DA47                 push    0               ; c
.text:0804DA49                 push    [ebp+buf]       ; s
.text:0804DA4C                 call    _memset
.text:0804DA51                 add     esp, 10h
.text:0804DA54                 sub     esp, 4
.text:0804DA57                 push    64h             ; n
.text:0804DA59                 push    0               ; c
.text:0804DA5B                 lea     eax, [ebp+cmd]
.text:0804DA5E                 push    eax             ; s
.text:0804DA5F                 call    _memset
.text:0804DA64                 add     esp, 10h
.text:0804DA67                 lea     eax, [ebp+cmd]
.text:0804DA6A                 mov     dword ptr [eax], 6F686365h
.text:0804DA70                 mov     word ptr [eax+4], 20h
.text:0804DA76                 lea     eax, [ebp+cmd]
.text:0804DA79                 add     eax, 5
.text:0804DA7C                 sub     esp, 8
.text:0804DA7F                 push    [ebp+str]       ; src
.text:0804DA82                 push    eax             ; dest
.text:0804DA83                 call    _strcpy
.text:0804DA88                 add     esp, 10h
.text:0804DA8B                 sub     esp, 0Ch
.text:0804DA8E                 push    [ebp+str]       ; s
.text:0804DA91                 call    _strlen
.text:0804DA96                 add     esp, 10h
.text:0804DA99                 lea     edx, [eax+5]
.text:0804DA9C                 lea     eax, [ebp+cmd]
.text:0804DA9F                 add     eax, edx
.text:0804DAA1                 mov     dword ptr [eax], 6D207C20h
.text:0804DAA7                 mov     dword ptr [eax+4], 75733564h
.text:0804DAAE                 mov     word ptr [eax+8], 6Dh
.text:0804DAB4                 sub     esp, 8
.text:0804DAB7                 push    offset aR_1     ; "r"
.text:0804DABC                 lea     eax, [ebp+cmd]
.text:0804DABF                 push    eax             ; command
.text:0804DAC0                 call    _popen
.text:0804DAC5                 add     esp, 10h
```

Figure 9. *strcpy()* call on *get_MD5()* function.

To test the security issue, it is necessary to input a large input on the password field during the iSQL's execution.



```
mphx2@kali:~/isql/iSQL-master$ ./isql

*************** welcome to ISQL ****************
* version 1.0                                  *
* Designed by RL                               *
* Copyright (c) 2011, RL. All rights reserved  *
************************************************

>username:test
>password:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Figure 10. Testing the application with a long password.

The "Segmentation fault" message on Figure 10 confirms the bug. For making sure that the issue is *exploitable* – if it can be used for compromising the system – It is necessary an error

[4] https://github.com/roselone/iSQL

debugging where all the buffer overflow's aspects are covered including the affected registers and variables.

The iSQL tool is a small software that could be easily evaluated, although the concepts applied on this case study can be extended to any other binary file.

A. Preventive Actions

There are techniques that can be applied during the software development that hamper the binary disassembling. The anti-disassembling techniques are mainly done using packing with some data compression algorithm as UPX[5] or employing techniques to confuse the execution and the understanding. For example, using many unnecessary loops or executing cheat codes [10].

V. Related work

Decompilers [11] are an alternative to retrieve source-code from the binary. Utilizing this kind of tools, a researcher can extract information to mount a *possible* code.

While possessing the source-code, a review can be done to make sure that common mistakes is being avoided on the development, issues like freeing two times the same memory allocation or not freeing at all, is something easy to detect during the code review and maybe not so obvious on the disassembling. Although, even with the source-code, this task can be tedious due to the code extension or an unusual development language.

Still, a decompiler sometimes cannot be able to extract the software's source-code because anti-decompiling techniques like obfuscation[6] for example. And even with anti-obfuscation tactics, the code will barely reflect the original source-code after the *deobfuscation*, which raises the difficulty of the assessment [12].

Decompiling technique complements the disassembling and it also should be used, if possible, to decrease the risk to a critical vulnerability on the assessed application.

VI. Final Remarks

There are multiple approaches for identifying security issues on software and they are very appreciated in this area. Even with all the available resources, it is a constant challenge to develop a safe application.

Assess an Assembly code is not an easy process, requires time and expertise. However, after consolidating the information extracted from the code, it is possible to understand the execution and being able to detect issues even with no source-code available. This paper demonstrates this resource for vulnerability detection and should be always utilized as assessment by researchers.

For a better interaction with the disassembled application, there is the intermediary language (IL) – each disassembler has its own, that is used for abstracting the Assembly code: reducing the functions number and deducing the results. It transforms the Assembly code into a more feasible language and in some cases, also enables the ability to work with it. Binary Ninja[7] is an excellent tool for this matter, allowing the researcher to develop with the binary using Python and its own API. Ultimately, the researchers intend to utilize IL, as a resource, for detecting different vulnerabilities like *Use-After-Free* for example, where an allocated memory is freed however the original pointer still points to some place in the same allocation, corrupting the validly used data and then changing the execution flow's behavior[8].

Future works include analyzing recent vulnerabilities to discover the real effect that their exploitation could have. For example, if a vulnerability considered only *Denial of Service* could turn into a *Remote Code Execution*, etc. Also develop a fuzzing framework to work with symbolic execution and grammar for identifying new security issues on modern browsers. This approach involves symbolic execution to discover paths on the application for possible vulnerable functions. The inputs are created to satisfy these paths which becomes the *fuzzer* more efficient [13] using less resources for fuzzing with better results – finding new vulnerabilities.

References

[1] Milsom, F. D. Why reverse engineering software? Reverse Engineering for Software Based Systems, IEEE Colloquium on, London. pp. 1/1-1/3, 1994.

[2] Eilam, E. Reversing – Secrets of Reverse Engineering, 1d. John Wiley Consumer, 2005.

[3] Anley, C. Heasman, J. Lindner, Felix and Richarte, G. The Shellcoder's Handbook, 2d. Wiley Publishing, Inc, 2007.

[4] Weigand, K. A. and Hartung, R.. Abduction's role in reverse engineering software. Aerospace and Electronics Conference (NAECON), 2012 IEEE National, Dayton, OH, pp. 57-6, 2012.

[5] Kaminsky, D. Reverse Engineering Code with IDA Pro, 1d. Syngress, 2008.

[6] Oliveira. B. Browser White Box Fuzz Testing: Um Estudo de Caso Sobre Detecção de Vulnerabilidades Utilizando Execução Simbólica. Proposta de Dissertação de Mestrado, UTFPR-CP, 2016.

[7] Felmetsger, V. Cavedon, L., Kruegel, C. and Vigna, G. Toward automated detection of logic vulnerabilities in web applications. USENIX Security'10 Proceedings of the 19th USENIX conference on Security Pages 10-10, 2010.

[8] Erickson, J. Hacking: The Art of Exploitation, 2d. No Starch Press, 2008.

[9] Li, C. Wei, Q. and Wang, Q. RankFuzz: Fuzz Testing Based on Comprehensive Evaluation. Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference. 939-942, 2012.

[10] Gangnon, M. Taylor, N. S. and Ghosh, A. K. Software Protecting Through Anti-Debugging. IEEE Security & Privacy, vol. 5, no. 3, 2007.

[11] Guilfanov, I. Decompilers and beyond, 2008.

[12] S.K. Udupa, S.K. Debray and M. Madou. Deobfuscation: reverse engineering obfuscated code. Reverse Engineering, 12th Working Conference, 2005.

[13] Stephens, N. et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution, 2016

[5] http://upx.sourceforge.net/

[6] https://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-sean_taylor-binary_obfuscation.pdf

[7] https://binary.ninja/

[8] http://cwe.mitre.org/data/definitions/416.html