

VULNSCAN 2022



7 JANVIER

VulnScan

By Raph and Joss



Vulnerability discovery in binary, plugin for IDA

**Mapping of potentially dangerous with their path
from potential user-controlled input**

Names of participants: Joss Lalou / raphael ohana

Name of teacher : Arie haenel

Year :5782



SUMMARY

Introduction.....	4
I - Definition of Problem.....	5
II - Architecture.....	7
III - IDAPython.....	8
3.1 import and functions.....	8
3.2 class function	10
3.3 Class Tree	13
IV - Trace Back	15
4.1 32 bits	15
4.2 64 bits	20
V - Analyze of functions.....	22
VI - Use the tool	24
VII - Limit of the tool	27
VIII - Summary diagram	29
conclusion.....	30
references.....	31

Introduction

Abstract : Today, we live in a generation where the world is evolving, but evolving more and more quickly, which says evolution also means greater vulnerabilities/threats , by virtue of his nature man tends to want to control everything, whether in his life, in his work, and especially if this can cause him harm, we tend to want to control so-called 'unforeseen' cases outside the limits initially foreseen, what is the relationship with the chosen subject? It is in truth obvious: any open source, which portion of code, which project even the most developed can make use of functions named: “functions input”, which require the intervention of the user, it is as you could imagine the danger, our mission? To create a tool analyze for Ida: we research in the exe code of a lambda code and find what possible threats /vulnerabilities related to the input are present here. It is necessary to develop techniques that can detect vulnerabilities automatically.

We know that there are many vulnerabilities present due to (or thanks to :-) input functions, such as scanf , gets, memset ... These same functions affect the rest of the code and cause flaws in other types of string processing functions: strcpy , strcmp , strlen ...

The question is:

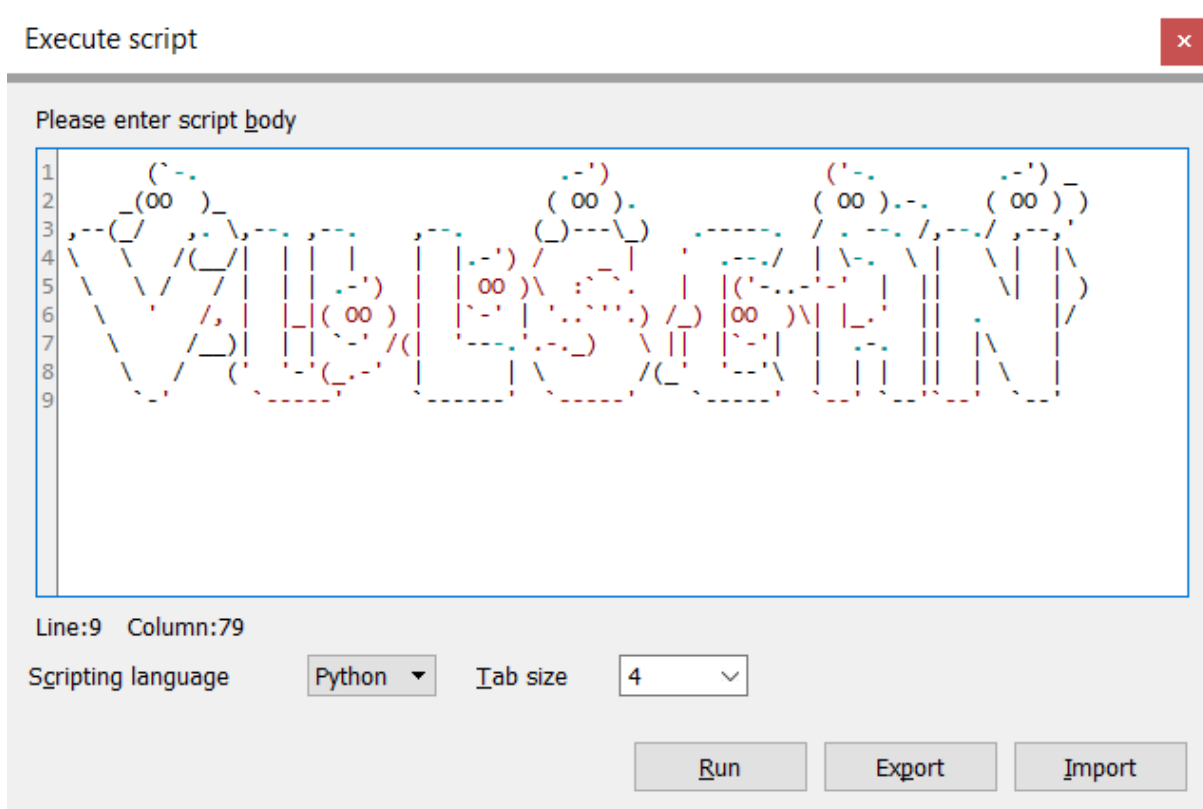
How can we propose a solution not to fix the vulnerabilities, but to accompany the vulnerability research on Ida?

I - Definition of the Problem

All the work done has been under IDA, as explained in class it is an interactive disassembler with many more features compared to the more usual disassemblers.

Reminder: Disassembly is a big part of security research. It will help you dissect that Microsoft patch to discover the silently fixed bugs they don't tell you about, or more closely examine a server binary to determine why your exploit isn't working. Many debuggers are available, but IDA Pro has become the de-facto standard for the analysis of hostile code and vulnerability research.

In IDA, there is a very useful feature that allows you to use a python script and run it on the code in assembly: IDAPython.



In this presentation we will begin to describe using basic IDAPython techniques to detect dangerous programming constructs which often result in stack-buffer overflows/override or format string attack.

We will go into more detail throughout this report, but the goal is: to use a python script in IDA to scan assembly code and to repair functions defined as "vulnerable", to trace their parameters in order to discover their possible flaws.

While the number of vulnerabilities is increasing rapidly, the response to them relies on manual analysis, resulting in a slow response speed.

To solve this problem, we have created **VULNSCAN**.

VulnScan is a fantastic tool that can be used to automate tedious or complicated reverse engineering tasks. We are using this tool to assist in auditing binaries for vulnerabilities. Since this is not a new idea (Halvar Flake presented on automating vulnerability research with IDA scripting in 2001), this may be partially a result of the increasing complexity required to perform exploitation on modern operating systems. However, there is still a lot of value in being able to automate portions of the vulnerability research process.

II - Architecture

All our project is based on the analysis of vulnerable function to prevent of a “user-controlled input that could trigger a bug”.

This analysis is build in many parts, we need to find all the functions from the file, find the parameters, trace back theirs values, etc....

As you can imagine we have a lot of data to store, all the functions with all their properties.

We decided to create the architecture of the code like this:

- First, we get all the functions of the executable.
- From this list we sort all the vulnerable functions that we decide (you will see later).
- We trace back the parameters of each vulnerable function.
- We define a tab of “mother-function”, in this tab we have all the function where a vulnerable function is in.
- In order to store all the data, we create a class Function with all the properties we need so each function that will be store in a tab will be an object from the Function type.
- We will use a Tree as a data structure where each node is an object Function from the class Function.
- To scan the vulnerability, we will run in the tree from vulnerable function to the head of the tree.

III – IDAPython

3.1) import and functions

All the project wad coded in python as a script for IDA pro:

A lot of import was used:

I will explain all the imports and all the special functions used in the code.

```
import anytree
import idutils
import idc
import idaapi
import ida_xref
import copy
import ida_kernwin
```

AnyTree: ([link in ref.](#))

From this package we can create a Tree data structure with a lot of possibilities:

We used:

- anytree.search.find(head,lambda node : node.name==item[1])

Return the node where the condition is respected.

- anytree.search.findall(head,lambda node : node.name==item[1])

Return the nodes where the condition is respected.

Idautils:

We used:

- idautils.Functions()

Return a list of address of all the functions in the executable file.

- idautils.CodeRefsTo(addr, False)

Return a list of all the "Code reference To" (where this function is called) , the param addr given in the function is any address in the function we want to know where she was called.

- idautils.StructMembers(frame)

Return a dictionary of the stack of a function. The frame given in param is a frame of the function.

(see IDC to get the frame)

Idc:

We used:

- idc.get_func_name(addr)

Return the name of the function where this addr come from.

- idc.get_func_attr(addr, idc.FUNCATTR_START)

Return the address of the head of the function (address of : push ebp)

- idc.print_insn_mnem(addr)

Return the instruction of the line in this address (0x00401000 mov eax,0) return mov

- idc.print_operand(prev_addr, 0)

Return the operand number 0 of the line in this address (0x00401000 mov eax,ebx) return eax.

- idc.GetDisasm(addr)

Return the line of this address ex : `xor eax,eax`

- `idc.get_frame_id(func.addr)`

Return the frame a function by sending an address as parameter

Idaapi:

We used:

- `idaapi.get_inf_structure()`

Return the structure of the file (64bit/32bit)

ida_kernwin:

We used:

- `ida_kernwin.ask_yn(0,"Welcome/n Are you ready ?")`

Return the Yes/No MessageBox

Copy:

We used:

- `.copy()`

Return shallow copies. (ex: `my_list.copy()`)

- `copy.deepcopy()`

Return a deep copy of x

3.2) class function

We decided to work in OOP, we build a class Function that will allow us to store every data we get from IDA pro.

```
class Function:
```

```
##### constructor #####
```

```
    def __init__(self,addr,name):
        self.addr =addr
        self.name = name
        self.xref_tab = []
        self.dict_arg_adress = {}
        self.dict_arg_value = {}
        self.dict_stack={}
        self.dict_in_function={}
        self.stack_base=0

        self.trace_back()
```

Here all the properties:

- **Name** : the name of the functions
- **Addr** : the address of the function
- **Xref_tab**: a tab that contains all the address where this function was called
- **Dict_arg_adress**: a dictionary fulfill with the register passed to the function (before trace back), the key is the address, the value is the register.
- **Dict_arg_value**: a dictionary fulfil with the value of register passed to the function (after trace back), the key is the address, the value is the value of the register.
- **Dict_stack**: a dictionary of the function's stack.
the key is the place of the param in the stack the value is the name of the param.

- **Dict_in_function**: a dictionary of the “mother-function” (function in which I am)

Function:

- **Trace_back()**:

This function will call the correct trace back for the function(64bit / 32bit)

Examples:

For this part of code:

```
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      sub     esp, 14h
.text:00401009      nop
.text:0040100A      mov     eax, 0
.text:0040100F      mov     [ebp+var_4], eax
.text:00401012      mov     eax, [ebp+Source]
.text:00401015      push    eax           ; Source
.text:00401016      lea     eax, [ebp+Destination]
.text:00401019      push    eax           ; Destination
.text:0040101A      call    strcpy
```

The result:

```
name of the function:      strcpy
adress of the function:    0x40101a
in the function           {'0x401000': 'sub_401000'}
adress of the xref_to      []
dict of the param-addr     {'0x401019': 'eax', '0x401015': 'eax'}
dict of the param-value    {'0x401016': 'ebp+Destination', '0x401012': 'ebp+Source'}
dict of stack              {}
base of the stack          0
```

3.3) Class Tree

As explained we will use a Tree to structure our data:

```
class Tree(Function, anytree.NodeMixin):
    def __init__(self, name, fun, parent=None, children=None):
        self.function = fun
        self.name = name
        self.parent = parent
        if children: # set children only if given
            self.children = children
```

The class Tree will create a Node. In each node there are some properties:

- **function**: the object function from the class Function
- **name**: the name of the node
- **parent**: the parent Node
- **children**: the children Node

An example of a tree we created from the Lab8 /

auth_overflow.exe:

You have the code in language C and the Tree we build with the tool. As you can see we have many printf in the main,

We have strcpy and strcmp in check_authentication and this one is in main too.

The function start will appear in IDA pro.

```

int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "jctrocks") == 0)
        auth_flag = 1;

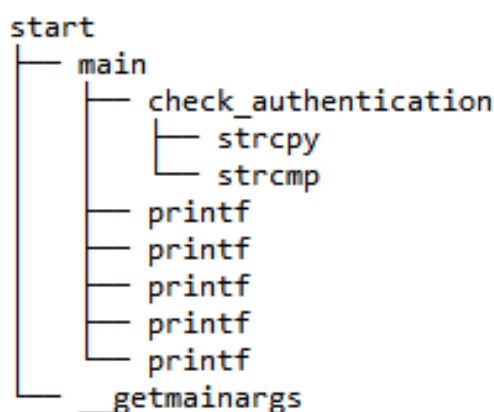
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }

    if(check_authentication(argv[1])) {
        printf("\n-----\n");
        printf("        Access Granted.\n");
        printf("-----\n");
    } else {
        printf("\nAccess Denied.\n");
    }
}

```

____TREE OF THE IDA VULNERABLE FUNCTION____



As you can see the leaves of the Tree are vulnerable functions, with this kind of Tree we can have a global picture of the code very quickly.

IV - Trace Back

One of the main difficulties of the tool was the trace back of the parameters of each function:

In assembly how do you write a function? how do you pass the parameters?

First, we cut the trace back in 2 parts:

The 32 bit part, and the 64 bit part.

4.1) 32 bits

In 32 bit all the parameters are given to the stack with the instruction push:

The order of the push is the inverse of how you read the function written with a high language :

In language C:

- `my_func(int a, int b, int c)`

In language assembly:

```
push c
push b
push a
call my_func
```

how do we trace back ? :

```
check_authentication proc near
```

```
Destination= byte ptr -14h
```

```
var_4= dword ptr -4
```

```
Source= dword ptr 8
```

```
push    ebp |
mov     ebp, esp
sub     esp, 14h
nop
mov     eax, 0
mov     [ebp+var_4], eax
mov     eax, [ebp+Source]
push    eax ; Source
lea     eax, [ebp+Destination]
push    eax ; Destination
call    strcpy
```

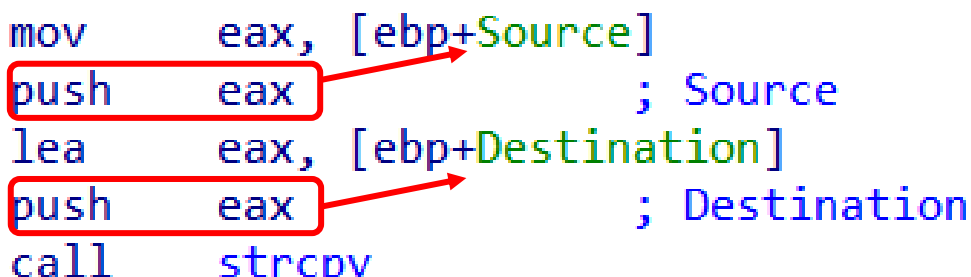
Here I want to trace back the param of the strcpy function:

First I store the push with theirs address in the

function.dict_arg_adress:

After that I will trace back the value of each push:

```
mov     eax, [ebp+Source]
push    eax ; Source
lea     eax, [ebp+Destination]
push    eax ; Destination
call    strcpy
```



After getting the value, you need to know where it was defined in the stack , defined as a local variable or defined as an argument that was sended to the “mother-function”.

Here the head of the function in IDA pro:

```
check_authentication proc near
```

```
Destination= byte ptr -14h  
var_4= dword ptr -4  
Source= dword ptr 8
```

As seen in the stack of the check_authentication function:

Destination = -14h, it is a negative value that means that I take some space in my stack to store this variable so it's a local variable defined IN the function check_authentication.

Source = 8, it is a positive value that means that I don't take some space in my stack, but I will receive this arg as a push before the call of check_authentication.

The new challenge now is to trace back also this type of variable, variable not local.

So what we did ? :

I search if my var is local or not, if it's not:

I search in the stack what is the index of the argument.

In our case we just have one variable with a positive value Source, his index is 1.

I will trace back the param number 1 of the function check_authentication:

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 0
nop
mov     eax, [ebp+argc]
cmp     eax, 2
jge     loc_401087

p+argv]
x]

set Format ; "Usage: %s <password>\n"
; Format

loc_401087:
mov     eax, [ebp+argv]
add     eax, 4
mov     ecx, [eax]
push    ecx ; Source
call    check_authentication
```

Here I trace back the first push above the call, the value is argv .

Here the value of argv is positive so the var come from the “mother-function” of main.

I get the index of argv in the stack of main (who is the mother-function of check_authentication), I get 2 , so I will trace back the 2nd push before the call main:

```
call    __getmainargs
add     esp, 14h
mov     eax, [ebp+envp]
push    eax ; envp
mov     eax, [ebp+argv]
push    eax ; argv
mov     eax, [ebp+argc]
push    eax ; argc
call    main
```

The stack of the function start where the function main is in :

```
public start
start proc near

var_14= dword ptr -14h
Code= dword ptr -10h
envp= dword ptr -0Ch
argv= dword ptr -8
argc= dword ptr -4
```

Here argv as a negative value so I can conclude that argv come from the cmd line.

Exemple of strcpy in check in main:

name of the function:	strcpy
adress of the function:	0x40101a
in the function	{'0x401000': 'check_authentication'}
adress of the xref_to	[]
dict of the param-addr	{'0x401019': 'eax', '0x401015': 'eax'}
dict of the param-value	{'0x401016': 'ebp+Destination', '0x401012': 'ebp+Source'}
dict of stack	{}
base of the stack	0
name of the function:	check_authentication
adress of the function:	0x401090
in the function	{'0x40104f': 'main'}
adress of the xref_to	[]
dict of the param-addr	{'0x40108f': 'ecx'}
dict of the param-value	{'0x401087': 'ebp+argv'}
dict of stack	{0: 'Destination', 16: 'var_4', 20: ' s', 24: ' r', 28: 'Source'}
base of the stack	24
name of the function:	main
adress of the function:	0x40113e
in the function	{'0x4010e0': 'start'}
adress of the xref_to	[]
dict of the param-addr	{'0x40113d': 'eax', '0x401139': 'eax', '0x401135': 'eax'}
dict of the param-value	{'0x40113a': 'ebp+argc', '0x401136': 'ebp+argv', '0x401132': 'ebp+envp'}
dict of stack	{0: ' s', 4: ' r', 8: 'argc', 12: 'argv', 16: 'envp'}
base of the stack	4

4.2) 64 bits

In 64 bit all the parameters are not given to the stack:

The 4 firsts parameters are passed in the registers: Rcx, Rdx, R8, R9.
if they are more than 4 parameters, the others will be send in the stack like in 32bit.

The idea of the trace back is the same:

The function properties will be fullfil in the order of the trace back :
First rdx and after rcx.

Here the targuil “ex1_ShmouelAbergel.exe” from the lab1 2021:

```
lea    rax, aEnterYourGuess ; "enter your guess ?\n"
mov    r11, rax
lea    rax, Format           ; "%s"
mov    r10, rax
mov    rcx, r10              ; Format
mov    rdx, r11
call   printf
lea    rax, [rbp+var_6]
mov    r11, rax
lea    rax, aS               ; "%s"
mov    r10, rax
mov    rcx, r10              ; Format
mov    rdx, r11
call   scanf
```

After finding the register given (rdx and rcx)

We trace back the value of each register:

Scanf function, exemple :

Rdx > r11 > rax >[rbp+var_6]

Exemple on scanf in the function sub_4010D0:

```
name of the function:      scanf
adress of the function:    0x401111
in the function           {'0x4010d0': 'sub_4010D0'}
adress of the xref_to      []
dict of the param-addr    {'0x40110e': 'rdx', '0x40110b': 'rcx'}
dict of the param-value   {'0x4010fa': 'rbp+var_6', '0x401101': 'a$; "%s"'}
dict of stack             {}
base of the stack         0
```

And printf also in sub_4010D0:

```
name of the function:      printf
adress of the function:    0x4010f5
in the function           {'0x4010d0': 'sub_4010D0'}
adress of the xref_to      []
dict of the param-addr    {'0x4010f2': 'rdx', '0x4010ef': 'rcx'}
dict of the param-value   {'0x4010db': 'aEnterYourGuess; "enter your guess ?\\n"', '0x4010e5': 'Format; "%s"'}
dict of stack             {}
base of the stack         0
```

Sub_4010D0 in start:

```
name of the function:      sub_4010D0
adress of the function:    0x4012b5
in the function           {'0x401200': 'start'}
adress of the xref_to      []
dict of the param-addr    {'0x4012b2': 'rdx', '0x4012af': 'rcx', '0x40129d': 'r8'}
dict of the param-value   {'0x401288': 'cs: __argv', '0x4012a6': 'rbp+var_18', '0x40128f': 'cs: _environ'}
dict of stack             {42: 'var_6', 43: 'var_5', 44: 'var_4', 45: 'var_3', 46: 'var_2', 48: ' s', 56: ' r'}
base of the stack         56
```

Start function :

```
name of the function:      start
adress of the function:    0x401200
in the function           {}
adress of the xref_to      []
dict of the param-addr    {}
dict of the param-value   {}
dict of stack             {32: 'var_30', 56: 'var_18', 64: 'var_10', 76: 'var_4', 80: ' s', 88: ' r'}
base of the stack         88
```

The start function as lot of properties empty because it is the mother of the function ther is no function that call her etc...

V - Analyze of functions

Functions we decided to analyse:

Scanf:

```
int scanf(const char *format, ...)
```

This function can have many parameter , the first is always the format.

we will analyse the first parameter in order to know if it the format was rightly written. And the second in order to know if this arg is local (for a stack overflow) or where he come from.

Scanf_s:

```
scanf_s("%your_format", buffer, (unsigned)_countof(buffer));
```

scanf_s has an argument(parameter) where you can specify the buffer size and actually control the limit of the input so you don't crash the whole building.

We decided to analyse the format and the arg not the counter.

Gets:

```
char *gets(char *str)
```

str – This is the pointer to an array of chars where the C string is stored.

This function returns str on success, and NULL on error or when end of file occurs, while no characters have been read.

Here as the scanf we want to know id the str is in the stack somewhere (in the local stack or not)

Printf:

```
int printf(const char *format, ...)
```

This function can have many parameter , the fisrt is always the format.

we will analyse the first parameter in order to know if it the format come from a user controlled input.

Strcpy:

```
char *strcpy(char *dest, const char *src)
```

This function has 2 parameter we will analyse the 2nd (src) in order to know if it was inputed by a user.

Strcmp:

```
int strcmp(const char *str1, const char *str2)
```

compares the string pointed to by str1 to the string pointed to by str2.

We will analyse the 2nd parameter in order to know if it was inputed by a user.

VI - Use the tool

Using this tool is **very easy**, you will need to download some package:


Pip install anytree

After download this you need to copy this file and past it in the Paths:

IDA Educational 7.6/python/3


IDA Educational 7.6/python/2

Ce PC > Windows (C:) > Programmes > IDA Educational 7.6 > python > 3

^	<input type="checkbox"/> Nom	Modifié le	Type	Tail
	 anytree	27/12/2021 20:06	Dossier de fichiers	

Copy the doc anytree

> Ce PC > Windows (C:) > Programmes > IDA Educational 7.6 > python > 3

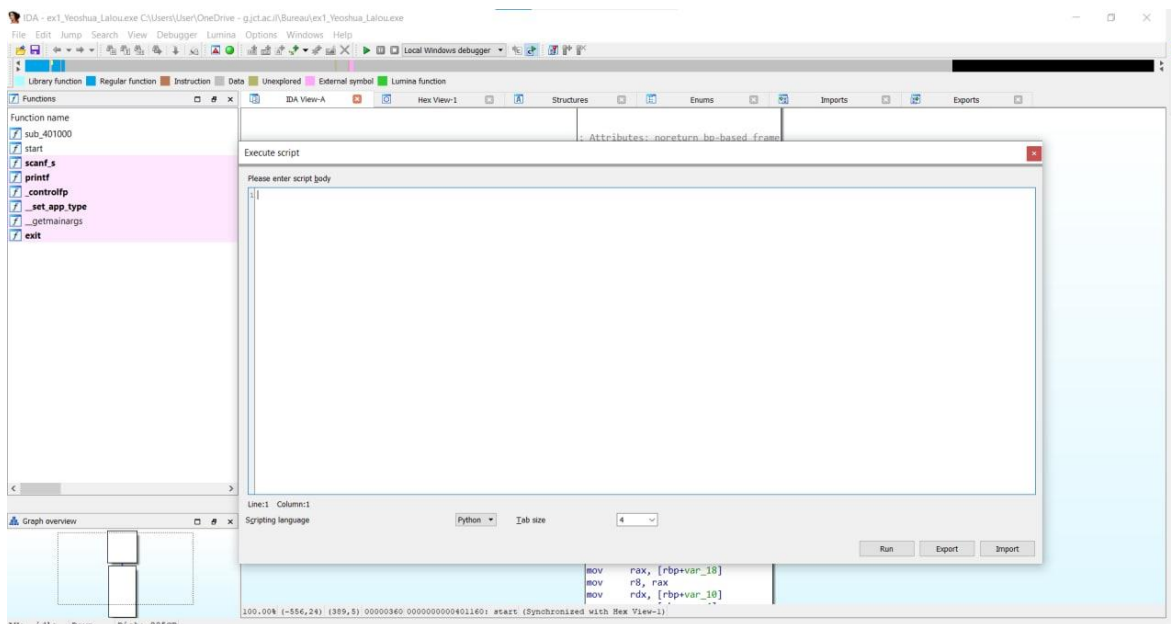
^	<input type="checkbox"/>  six.py Type : Python File	Modifié le : 27/12/2021 20: Taille : 33,7 Ko
---	---	---

Copy the file six.py

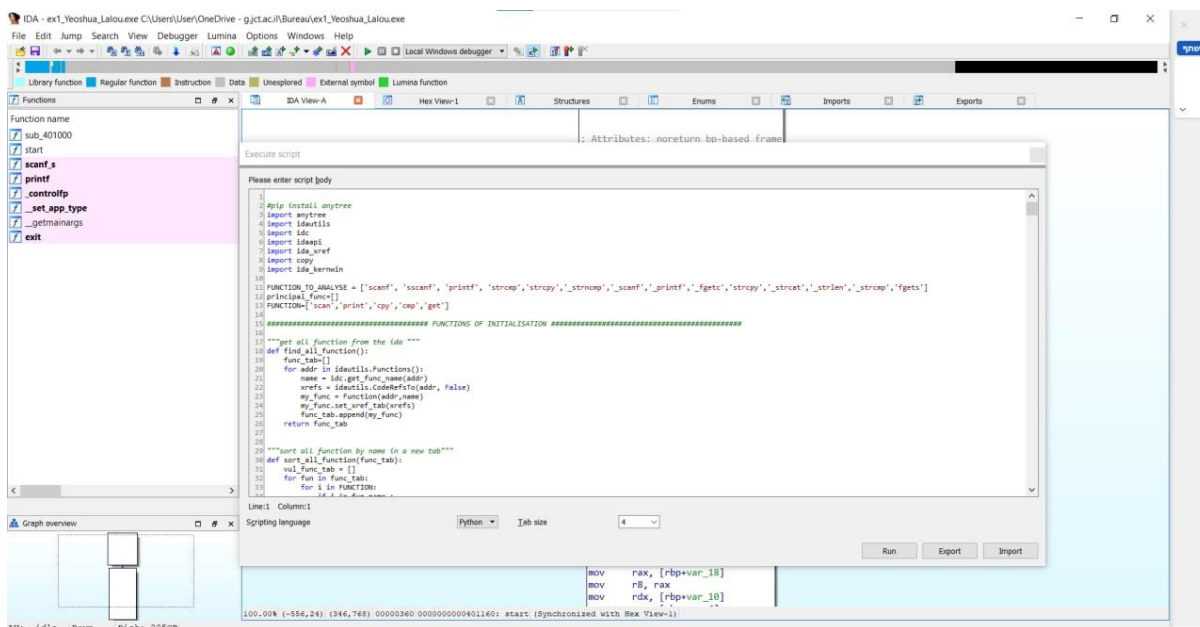
Place the plugin in a folder you want:

Open Ida

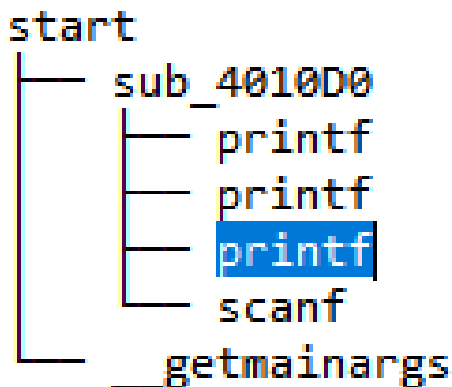
Click on file , script command:



Pass the language to python not IDC
Click on import and import the plugin:



All the function name in the tree are **clickable** click on it amazing no ?



All the address are also **clickable**:

```

name of the function:    printf
address of the function: 0x4010f5
in the function         {'0x4010d0': 'sub_4010D0'}
address of the xref_to  []
dict of the param-addr  {'0x4010f2': 'rdx', '0x4010ef': 'rcx'}
dict of the param-value {'0x4010db': 'aEnterYourGuess; "enter your guess ?\\n"', '0x4010e5': 'Format; "%s"'}
dict of stack           {}
base of the stack       0
  
```

If a vulnerability is discover you will see apers to the output screen of the ida console:

```

POSSIBLE VULNERABILITY OVERFLOW
scanf 0x401111 use the local var : var_6 (1) with a format non precise in the function sub_4010D0 0x4012b5|
  
```

Of course the address and the function name are also clickable:

We have 3 type of vulnerability:

- Overflow** (scanf/ scanf_s/gets/strcpy)
- Overread** (strcmp)
- Format String** (printf)

VII - Limit of the tool

1) Limit of the function chosen:

We limited or scan to some case in each function:

Scanf / scanf_s : we discover a vulnerability if the format is non precise (%s) and the first param after the format, we don't analyse the other params if there is more.

Strcpy / strcmp : we discover a "possible" vulnerability, we scan if the parameter given in the function is a user controlled input.

Printf : we scan all the printf function who have only one param.

2) Limit of the trace back:

We limited the trace back value to the closest call above the function I work on:

```
lea    rax, aEnterYourGuess ; "enter your guess ?\n"
mov    r11, rax
lea    rax, Format           ; "%s"
mov    r10, rax
mov    rcx, r10              ; Format
mov    rdx, r11
call   printf


---


lea    rax, [rbp+var_6]
mov    r11, rax
lea    rax, aS               ; "%s"
mov    r10, rax
mov    rcx, r10              ; Format
mov    rdx, r11
call   scanf
```

In the trace back of scanf, I will stop searching the value of a register if he was initialized before the closest call here the red line is the limit.

A limit is if a function is in a very low address in the code and use a register who was initialized at the beginning of the code, the trace back will not work.

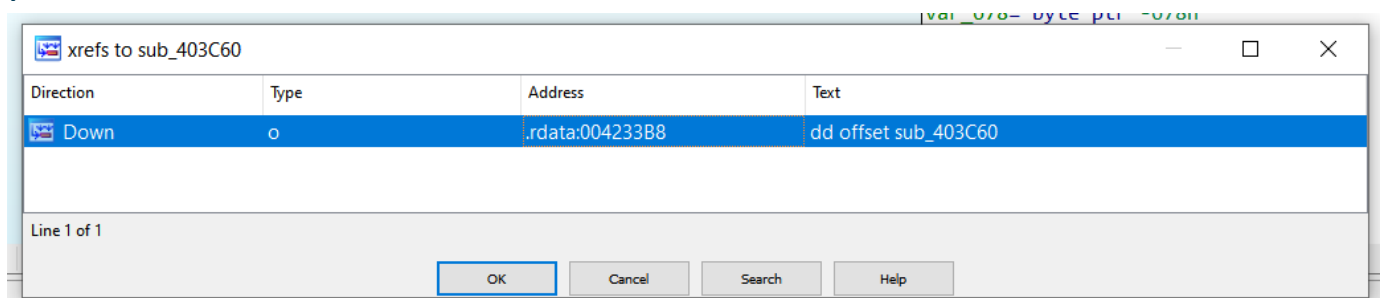
Traceback 64 bit:

We limit the trace back of the parameter to the local area , we will not trace back the parameter if he come from a “mother-function”.

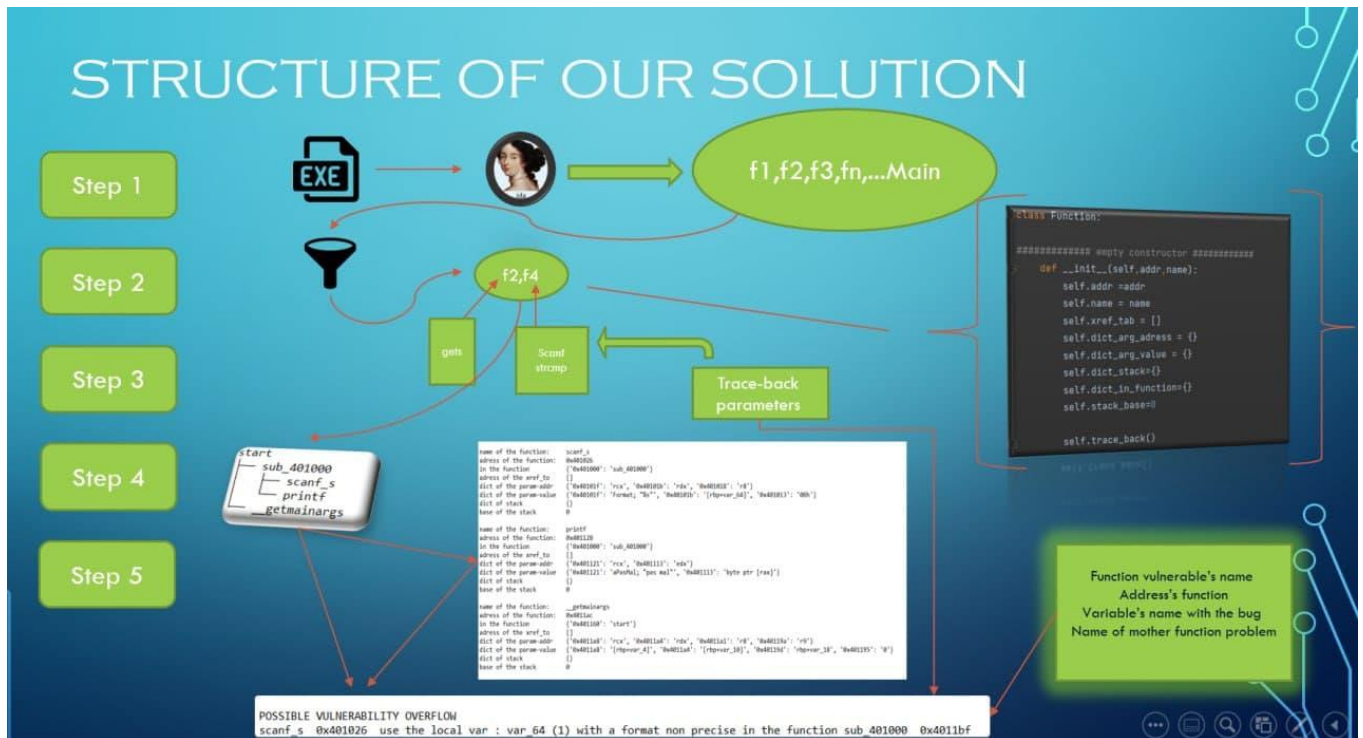
3) Limit of the tree

The tree was built with the xref of all function to know who call who.

Sometimes we have some function that are call in the .rdata , so the tree cannot add this function as a node because they don't have any parents



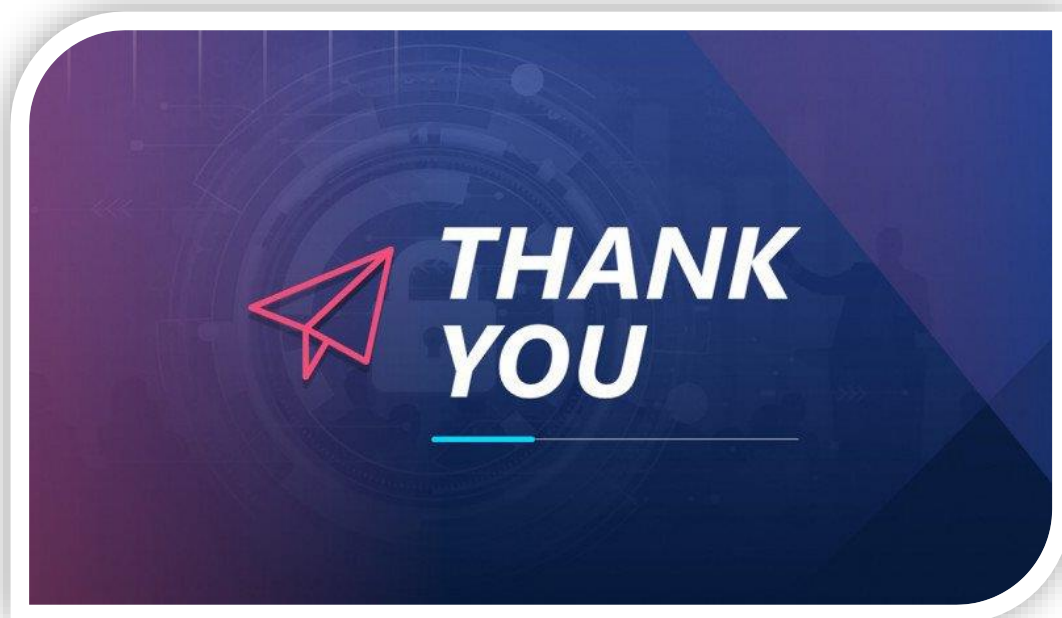
VIII - Summary diagram



Conclusion

The number of vulnerabilities is increasing rapidly due to the development of new hacking techniques. However, time-consuming software analysis depending on a vulnerability analyst make it difficult to respond to attacks immediately. We have defined our possible solution, of course we did the maximum in the time that was given to us, this tool has no development limit, our dream is to continue to develop, upgrade and update it.

Thank you for having had the chance to learn so many necessary keys whether for the world of work, for the army or more globally: for the world of cyber security.



References

anytree :

<https://anytree.readthedocs.io/en/2.8.0/>

copy :

<https://docs.python.org/3/library/copy.html>

lazylda :

<https://github.com/L4ys/LazyIDA/blob/master/README.md>

introduction to idapython :

<https://www.somersetrecon.com/blog/2018/7/6/introduction-to-idapython-for-vulnerability-hunting>

examples of codes vulnerables:

<https://github.com/snoopysecurity/Vulnerable-Code-Snippets>

scanf vulnerability :

<https://bigpick.github.io/TodayILearned/articles/2020-03/basic-scanf-vulnerability>

cwe buffer overflow :

<https://cwe.mitre.org/data/definitions/121.html>

idc for idapython :

<https://hex-rays.com/products/ida/support/idadoc/157.shtml>

idautils for idapython :

https://hex-rays.com/products/ida/support/idapython_docs/idautils.html

anytree for idapython :

<https://anytree.readthedocs.io/en/latest/>

idaapi for idapython :

https://www.programcreek.com/python/example/86101/idaapi.get_func

xrefs from idc for idapython :

<https://hex-rays.com/products/ida/support/idadoc/313.shtml>

ida kernwin for idapython :

https://www.programcreek.com/python/example/127892/ida_kernwin.Form

