Working folder for mona script

Command: !mona config -set workingfolder C:\ImmunityLogs\%p [Enter].

Mona script suggestion.

Command: !mona suggest [Enter].

Find jmp esp or call esp

Command: !mona jmp -r esp [Enter].

Command: !mona jmp -r esp -m kernel [Enter].

Command: !mona jmp -r esp -m \* [Enter].

Check ASLR

!mona modules [Enter].

!mona noaslr [Enter].

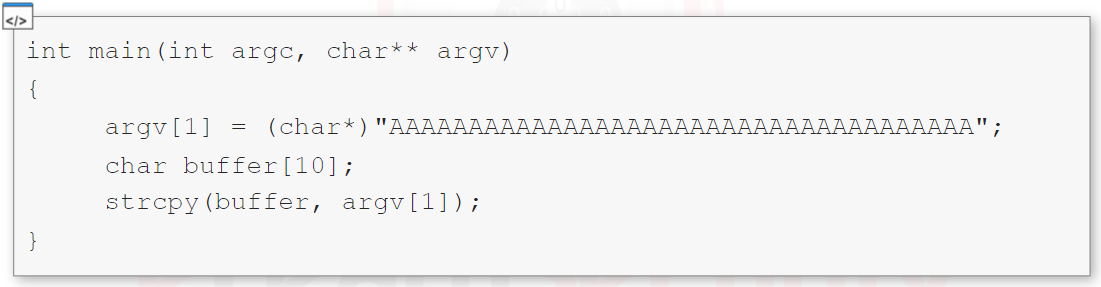
**Buffer Overflow**

**Buffer**

Buffer is loosely used to refer to any area in memory where more than one piece of data is stored. And **overflow** occurs when we try to fit more data than the buffer handle. You can think an overflow such as 5 gallons of water into a 4-gallon bucket.

[example 1 on buffer overflow]  
  
  


Suppose your last name is OVATALI (7 characters). Refusing to truncate your name, you write all 7 characters. The two extra characters have to go somewhere. This is where a buffer overflow happens.

[example 2 on buffer overflow]  


Code observation:

* The buffer size is 10 bytes long.
* The code use the function strcpy
  + strcpy function is used to copy value of one of variables to another.

Outcome:

We can see that argv[1] contains 35 A characters, while the buffer can handle only 10. When the preform runs, the exceeding memory: this is buffer overflow.

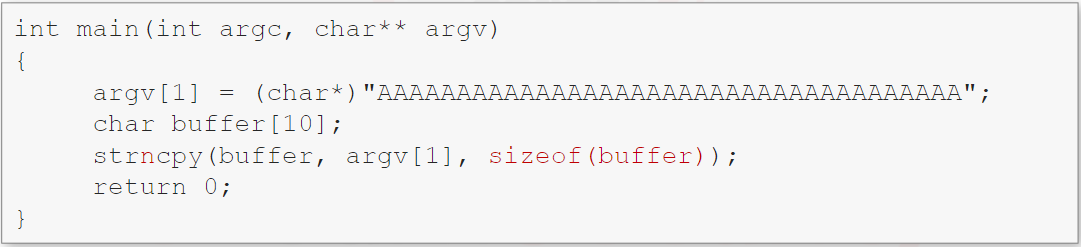
Code outcome:

Program crashes.

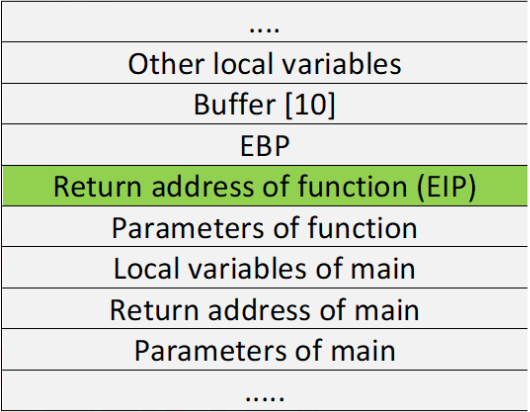
Post evaluation:

The vulnerable function is strcpy.

Resolution:



Let’s examine the above example by observing what is happening in the stack:

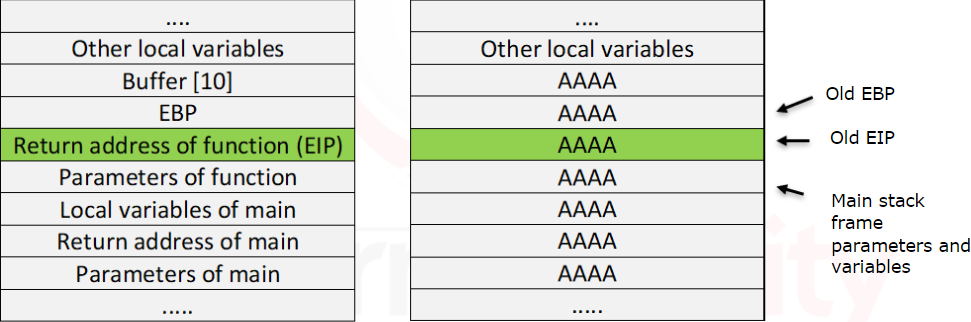


* push the function arguments.
* Call the function.
* Execute the prologue (which updates **EBP** and **ESP** to create the new stack frame).
* Allocate local variables.

When the strcpy function get executed, it start copying our input into the memory address allocated for buffer[10]. Since there is not enough space, our input will be copied in the next memory address and will continue to fil memory addresses until there is no more input.

What is getting overwritten?

An you can see in this stack presentation, this data includes the **EBP**, the **EIP** and all the other bytes related to the previous stack frame.



What can a pen tester do with this?

The **EIP** has been overwritten with AAAA, once the epilogue takes places, the program will try to return completely to wrong address. Remember that **EIP** points to the next instruction. An attacker can craft the payload in the input of the program to get the control of the program flow and return the function to a specific memory location. This is where it is important to know memory addresses of certain registers.

**Practical on buffer overflow**

STEP 1:

* Fuzzing input with random byte length with mona script for example.
  + Command: !mona pc <payload\_length> [Enter].
    - <payload\_length>: length of payload for example: 1200

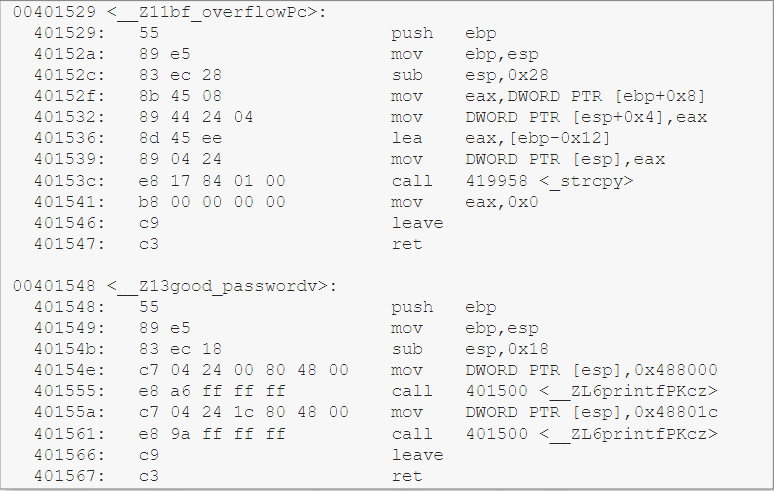
STEP 2:

* After application crashed, check the value of **EIP** and using mona script to identify the correct pattern offset.
  + Command: !mona po <eip\_value> <payload\_length> [Enter].
    - <eip\_value>: value of **EIP** after overwritten. Such as: 3251060
    - <payload\_length>: length of payload for example: 1200

**Note:** the previous step helping you to determine padding volume before byte that affect **EIP**.

STEP 3:

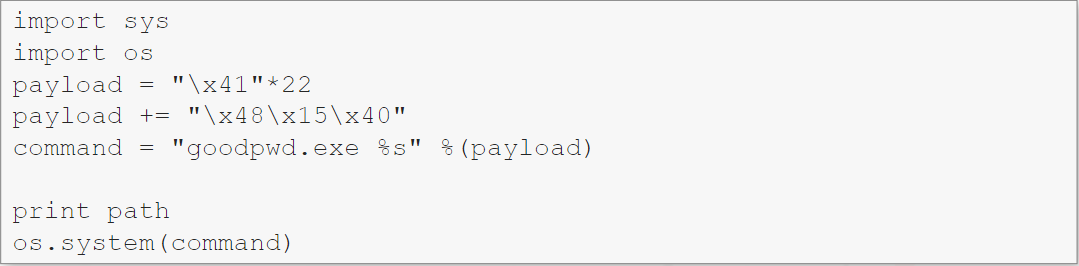
* Disassemble the program
  + Command: objdump -d -Mintel <application\_name.exe> > <filename.txt> [Enter].
    - <application\_name.exe>: executable file for disassemble.
    - <filename.txt>: identify the output file.

[example of output]  


**Note:** the previous step helping you to get function instruction address.

STEP 4:

* Preparing the write value for **EIP**.
  + Get instruction value from STEP 3.
  + Reverse instruction value in case of little-endianness.

[Example on payload]  


**Note:** notice that we did not add \x00 (Null Byte) to payload. You just need to know that when functions such as strcpy encounter a NULL BYTE in the source string, they will stop copying data.

We are successfully called overwrite **EIP** value with the desired function instruction value that mean we are able to change program’s flow.

**Note:** application could be crashed.

**Finding Buffer overflows**

There are unsafe operations, might be vulnerable to buffer overflows:

* strcpy function.
* strcat function.
* gets function / fgets function.
* scanf function / fscanf function.
* vsprintf function.
* printf function.
* memcpy function.

Any function which carries out the following operations may be vulnerable to buffer overflows:

* Does not properly validate input before operations.
* Does not check input boundaries.

**Note:** All the interpreted languages such as c#, Visual basic, .Net, JAVA, etc. are safe from such vulnerabilities.

Moreover, buffer overflows can be triggered by any of the following buffer operations:

* User input.
* Data loaded from a disk.
* Data from the network.

**Static analysis**

If you have access to the source code, there are tools such as splint, CppCheck, etc. such tools will try to detect not only buffer overflows but also some other types of errors.

* Splint tool download [link](http://www.splint.org/).
* CppCheck tool download [link](http://cppcheck.sourceforge.net/).

**Dynamic analysis**

When a crash occurs, be prepared to hunt for the vulnerability with a debugger (the most efficient and well-known technique).

* Fuzzer tool.
* Tracer tool.
* Cloud-fuzzing.

**Note:** All the above techniques can give a big number of vulnerabilities (such as overflows, negative indexing of an array and so on).

**Note:** A large number of vulnerabilities are un-exploitable. Almost 50% of vulnerabilities are not exploitable at all, but they may lead to DOS (denial of services attacks) or cause other side-effects.

**Fuzzing**

Fuzzing is a software testing technique that provides invalid data, i.e., unexpected or random data as input to a program, input can be in any form such as:

* Command line.
* Parameters.
* Network data.
* File input.
* Database.
* Shared memory regions.
* Keyboard/mouse input.
* Environment variables.

This technique basically works by supplying a random data to the program and then the program is checked for incorrect behavior such as:

* Memory hogging.
* CPU hogging.
* Crashing.

Note**:** it cannot be used to test all the cases.

Fuzzing tools and frameworks:

* Sulley tool download [link](https://github.com/OpenRCE/sulley).
* Sfuzz tool download [link](https://github.com/orgcandman/Simple-Fuzzer).
* FileFuzz tool download [link](http://packetstormsecurity.com/files/39626/FileFuzz.zip.html).
* Peach Fuzzing platform download [link](http://peachfuzzer.com/).

**Finding buffer overflow in binary programs**

STEP 1:

* We will go through the process of understanding to code then we will exploit it.
* Firstly, obtain the disassembled code using objdump.
  + Command: objdump -d -Mintel <app\_name.exe> > <output\_file.txt> [Enter].

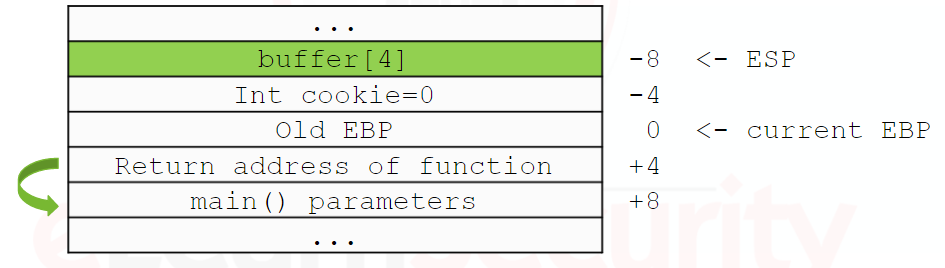
STEP 2:

* Finding entry point of the application.
  + 1st: searching on \_main at disassembled file.
  + 2nd: understanding the application flow by debugger or disassembled file.

Code observation

STEP 3:

* Since the function {gets} can be overflowed, we will demonstrate that we can:
  + Easily control program flow using variable control.
  + Buffer overflows can even be control via keyboard-inputs (through it’s hard to type shell-code using hand, but nonetheless, it can be done).
  + Find overflows in binaries.
* In our scenario, buffer[4]. How do we change the variable of previous variable that hardcoded in the code?



STEP 4:

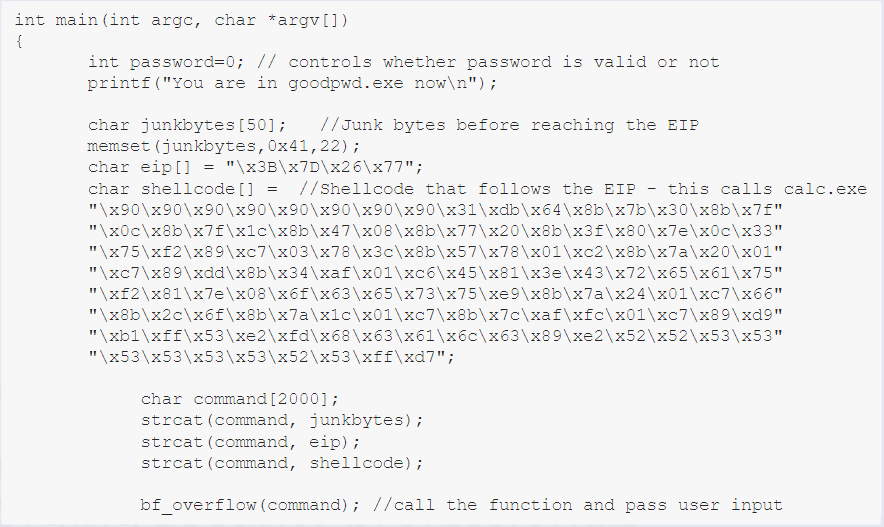
* Identified padding as 4 bytes.
* Overwrite on cookie=0 to be equal to right value to execute **CMP**.

STEP 5:

* We can do is find a **JMP ESP** (or **CALL ESP**) that is fixed location of memory.
* Trying to overwrite **EIP** with address of **JMP ESP** or **CALL ESP**.
* We could find **JMP ESP** or **CALL ESP** through:
  + Immunity debugger: CTRL+F. in the field, we will type **JMP ESP** or **CALL ESP** and then confirm.
  + Findjmp tool:
    - Command: findjmp.exe ntdll.dll esp [Enter].
  + Mona script:
    - Command: !mona jmp -r esp [Enter].
    - Command: !mona jmp -r esp -m kernel [Enter].
    - Command: !mona jmp -r esp -m \* [Enter].

STEP 6:

* After preparing padding value and identified JMP ESP or CALL ESP, we are going to identify shellcode.

[example on whole of payload]  


**Note:** NOPs (\x90) helping in sliding the **EIP** until reach out our shellcode.

